THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

ALGORITMOS

TEORIA E PRÁTICA

TRADUÇÃO DA
3ª EDIÇÃO AMERICANA

ELSEVIER

THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

ALGORITMOS

TEORIA E PRÁTICA

TRADUÇÃO DA
3ª EDIÇÃO AMERICANA

TRADUÇÃO
Arlete Simille Marques

REVISÃO TÉCNICA
Arnaldo Mandel
Departamento de Ciência da Computação
Instituto de Matemática e Estatística da USP

11ª tiragem

ELSEVIER

PREFÁCIO

Antes de existirem computadores, havia algoritmos. Mas, agora que temos computadores, há ainda mais algoritmos, e os algoritmos estão no coração da computação.

Este livro é uma introdução abrangente ao moderno estudo de algoritmos para computadores. Apresenta muitos algoritmos e os examina com considerável profundidade, embora torne seu projeto e sua análise acessíveis a leitores de todos os níveis. Tentamos manter as explicações em um nível elementar sem sacrificar a profundidade do enfoque ou o rigor matemático.

Cada capítulo apresenta um algoritmo, uma técnica de projeto, uma área de aplicação ou um tópico relacionado. Algoritmos são descritos em linguagem comum e em pseudocódigo projetado para ser fácil de ler por qualquer pessoa que tenha estudado um pouco de programação. O livro contém 244 figuras — algumas com várias partes — que ilustram como os algoritmos funcionam. Visto que enfatizamos a *eficiência* como um critério de projeto, incluímos análises cuidadosas dos tempos de execução de todos os nossos algoritmos.

O texto foi planejado primariamente para uso em cursos de graduação e pós-graduação em algoritmos ou estruturas de dados. Como discute questões de engenharia relacionadas ao projeto de algoritmos, bem como aspectos matemáticos, é igualmente adequado para profissionais técnicos autodidatas.

Nesta terceira edição, mais uma vez atualizamos o livro inteiro. As mudanças são abrangentes e incluem novos capítulos, revisão de pseudocódigos e um estilo de redação mais ativo.

Ao professor

Este livro foi projetado para ser ao mesmo tempo versátil e completo. Você descobrirá sua utilidade para uma variedade de cursos, desde graduação em estruturas de dados até pós-graduação em algoritmos. Como oferecemos uma quantidade consideravelmente maior de material da que poderia ser abordada em um curso típico de um período, você pode considerar o livro como um bufê de vários pratos do qual pode selecionar e extrair o material que melhor atender ao curso que deseja ministrar.

Você verá que é fácil organizar seu curso usando apenas os capítulos de que precisar. Os capítulos são relativamente autônomos, de modo que você não precisa se preocupar com uma dependência inesperada e desnecessária de um capítulo em relação a outro. Cada capítulo apresenta primeiro o material mais fácil e, em seguida, o material mais difícil; os limites das seções são pontos de parada naturais. Em cursos de graduação você poderia utilizar somente as primeiras seções de um capítulo; em cursos de pós-graduação, o capítulo inteiro.

Incluímos 957 exercícios e 158 problemas. Cada seção termina com exercícios, e cada capítulo com problemas. Em geral, os exercícios são perguntas curtas que testam o domínio básico do assunto. Alguns são exercícios simples de autoaferição, enquanto outros são mais substanciais e apropriados para o aluno resolver com mais tempo e calma. Os

problemas são estudos de casos mais elaborados que, muitas vezes, apresentam novo material; frequentemente consistem em várias perguntas que conduzem o aluno pelas etapas exigidas para chegar a uma solução.

Ao contrário da prática que adotamos em edições anteriores deste livro, nesta apresentamos soluções para alguns problemas, mas de modo algum para todos os problemas e exercícios. Essas soluções estão disponíveis no site da editora: www.elsevier.com.br/cormen. Seria interessante você visitar esse site para verificar se ele contém a solução para um exercício ou problema que planeja apresentar a seus alunos. Esperamos que o conjunto de soluções reunidos no site aumente ao longo do tempo, de modo que você deve visitá-lo toda vez que ministrar o curso.

Assinalamos com asteriscos (*) as seções e os exercícios mais adequados para alunos de pós-graduação do que de graduação. Uma seção marcada com asterisco não é necessariamente mais difícil que outra que não tenha asterisco, mas pode exigir o entendimento de matemática mais avançada. De modo semelhante, exercícios assinalados por asteriscos podem exigir um conhecimento mais avançado ou criatividade acima da média.

Ao aluno

Esperamos que este livro didático proporcione uma introdução agradável à área de algoritmos. Tentamos tornar cada algoritmo acessível e interessante. Para ajudá-lo quando encontrar algoritmos pouco familiares ou difíceis, descrevemos cada um deles etapa por etapa. Também apresentamos explicações cuidadosas dos fundamentos matemáticos necessários para entender a análise dos algoritmos. Se você já tiver alguma familiaridade com um tópico, perceberá que os capítulos estão organizados de modo que você possa apenas ler rapidamente as seções introdutórias e passar rapidamente para o material mais avançado.

Este é um livro extenso, e sua turma provavelmente só examinará uma parte de seu material. Porém, procuramos torná-lo útil para você, agora como livro didático, e também mais tarde, em sua carreira, como um guia de referência de matemática ou um manual de engenharia.

Quais são os pré-requisitos para a leitura deste livro?

- Você deve ter alguma experiência em programação. Em particular, deve entender procedimentos recursivos e estruturas de dados simples como arranjos e listas ligadas.
- Você deve ter alguma facilidade com demonstrações matemáticas, em especial por indução. Algumas partes do livro dependem de algum conhecimento de cálculo diferencial elementar. Além disso, as Partes I e VIII deste livro ensinam todas as técnicas matemáticas de que você necessitará.

Apresentamos soluções para alguns deles, que estão disponíveis em: www.elsevier.com.br/cormen. Você pode consultar o site e comparar suas soluções com as nossas.

Ao profissional

A ampla variedade de tópicos neste livro faz dele um excelente manual sobre algoritmos. Como cada capítulo é relativamente autônomo, você pode se concentrar nos tópicos que mais o interessem.

A maioria dos algoritmos que discutimos tem grande utilidade prática. Portanto, abordamos questões de implementação e outras questões de engenharia. Muitas vezes damos alternativas práticas para os poucos algoritmos cujo interesse é primordialmente teórico.

Se desejar implementar qualquer dos algoritmos, verá que a tradução do nosso pseudocódigo para a sua linguagem de programação favorita é uma tarefa razoavelmente direta. Projetamos o pseudocódigo para apresentar cada algoritmo de forma clara e sucinta. Consequentemente, não abordamos tratamento de erros e outras questões de engenharia de software que exigem características específicas do seu ambiente de programação. Tentamos apresentar cada algoritmo de modo simples e direto sem permitir que as idiossincrasias de determinada linguagem de programação obscureçam sua essência.

Se você estiver usando este livro por conta própria, sem seguir um curso, pode ser que você não consiga ter acesso às soluções de problemas e exercícios. No nosso site: **http://mi- tpress.mit.edu/algorithms/** há links para as respostas de alguns problemas e exercícios para que você possa verificar suas respostas. Por favor, não nos mande suas respostas.

Aos nossos colegas

Apresentamos indicações e bibliografia extensivas para a literatura corrente. Cada capítulo termina com um conjunto de notas do capítulo que dão detalhes e referências históricas. Contudo, as notas dos capítulos não oferecem uma referência completa para toda a área de algoritmos. Embora talvez seja difícil de acreditar, dado o tamanho deste livro, restrições de espaço nos impediram de incluir muitos algoritmos interessantes .

Apesar dos inúmeros pedidos dos alunos, preferimos manter a nossa política de não apresentar referências para as soluções de problemas e exercícios, para evitar que eles cedam à tentação de consultar uma solução dada em vez de determiná-la.

Mudanças na terceira edição

O que mudou entre a segunda e a terceira edições deste livro? Sobre a magnitude das mudanças entre essas duas edições e entre a primeira e a segunda, dizemos o mesmo que dissemos na segunda edição: dependendo do ponto de vista de cada leitor, a mudança pode não ser muito grande ou pode ser bem grande.

Um rápido exame do sumário mostra que a maior parte dos capítulos e seções da segunda edição aparecem na terceira edição. Eliminamos dois capítulos e uma seção, mas acrescentamos três novos capítulos e duas novas seções além desses novos capítulos.

Mantivemos a organização híbrida das duas primeiras edições. Em vez de organizar os capítulos só por domínios de problemas ou só de acordo com técnicas, este livro tem elementos de ambos. Contém capítulos baseados em técnicas de divisão e conquista, programação dinâmica, algoritmos gulosos, análise amortizada, NP-completude e algoritmos de aproximação. Mas também traz partes inteiras dedicadas a ordenação, estruturas de dados para conjuntos dinâmicos e algoritmos para problemas de grafos. Entendemos que, embora você precise saber como aplicar técnicas para projetar e analisar algoritmos, os problemas raramente informam de antemão quais técnicas são as mais adequadas para resolvê-los.

Damos a seguir um resumo das mudanças mais significativas para a terceira edição:

- Acrescentamos novos capítulos sobre árvores de van Emde Boas e algoritmos multithread, e agora a parte de fundamentos do material sobre matrizes ocupa um dos apêndices.
- Revisamos o capítulo sobre recorrências de modo a dar um tratamento mais abrangente à técnica de divisão e conquista, e suas duas primeiras seções aplicam essa técnica para resolver dois problemas. A segunda seção desse capítulo apresenta o algoritmo de Strassen para multiplicação de matrizes, que transferimos do capítulo sobre operações com matrizes.
- Eliminamos dois capítulos que raramente eram ensinados: heaps binomiais e redes de ordenação. Uma ideia fundamental no capítulo sobre redes de ordenação, o princípio 0-1, aparece, nesta edição, dentro do Problema 8-7 como o lema de ordenação 0-1 para algoritmos de comparação e permutação. O tratamento dos heaps de Fibonacci não depende mais do heaps binomiais como precursor.
- Revisamos o tratamento de programação dinâmica e algoritmos gulosos. Agora a programação dinâmica começa com um problema mais interessante, corte de hastes de aço, do que o problema de programação de linha de montagem na segunda edição. Além disso, enfatizamos a memoização com um pouco mais de intensidade do que o fizemos na segunda edição, e apresentamos a noção do grafo de subproblema como um modo de entender o tempo de execução de um algoritmo de programação dinâmica. Em nosso exemplo de abertura sobre algoritmos gulosos, o problema de seleção de atividades, chegamos ao algoritmo guloso mais diretamente do que o fizemos na

segunda edição.

- O modo de eliminar um nó em árvores de busca binária (que inclui árvores rubronegras) agora garante que o nó requisitado para eliminação seja o nó realmente eliminado. Nas duas primeiras edições, em certos casos, algum outro nó seria eliminado e seu conteúdo seria transferido para o nó enviado para o procedimento de eliminação. Com o nosso novo modo de eliminar nós, se outros componentes de um programa mantiverem ponteiros para os nós na árvore, não terminarão erroneamente com ponteiros inativos para nós que já foram eliminados.
- O material sobre redes de fluxo agora baseia os fluxos inteiramente em arestas. Essa abordagem é mais intuitiva do que o fluxo em rede usado nas duas primeiras edições.
- Como o material sobre fundamentos de matrizes e o algoritmo de Strassen passou para outros capítulos, o capítulo sobre operações com matrizes é menor do que o da segunda edição.
- Modificamos o tratamento do algoritmo de correspondência de cadeias de Knuth-Morris-Pratt.
- Corrigimos vários erros. A maioria deles aparece na errata da segunda edição publicada em nosso site, mas outros, não.
- Atendendo a muitos pedidos, mudamos a sintaxe (até certo ponto) do nosso pseudocódigo. Agora usamos "=" para indicar atribuição e "==" para testar igualdade, exatamente como fazem as linguagens C, C++, Java e Python. De modo semelhante, eliminamos as palavras-chave **do e then** e adotamos "//" como símbolo para comentários de final de linha. Agora também usamos a notação de ponto para indicar atributos de objetos. Nosso pseudocódigo continua sendo orientado a procedimento e não a objeto. Em outras palavras, em vez de executar métodos em objetos, simplesmente chamamos procedimentos e passamos objetos como parâmetros.
- Adicionamos 100 novos exercícios e 28 novos problemas. Além disso, atualizamos muitas citações bibliográficas e acrescentamos várias outras novas.
- Finalmente, repassamos o livro inteiro e reescrevemos sentenças, parágrafos e seções para tornar a linguagem mais clara e mais ativa.

Agradecimentos para a terceira edição

Estamos trabalhando com a MIT Press há mais de duas décadas, e que maravilhosa relação é a nossa! Agradecemos a Ellen Faran, Bob Prior, Ada Brunstein e Mary Reilly por sua ajuda e apoio.

Estávamos em lugares diferentes enquanto produzíamos a terceira edição, trabalhando no Dartmouth College Department of Computer Science, no MIT Computer

Science and Artificial Intelligence Laboratory e no Columbia University Department of Industrial Engineering and Operations Research. Agradecemos a nossas respectivas universidades e colegas por proporcionarem tais ambientes de suporte tão estimulantes.

Mais uma vez, Julie Sussman, P.P.A., nos salvou como editora técnica. Mais de uma vez ficamos pasmados com os erros que deixamos passar, mas que Julie percebeu. Ela também nos ajudou a melhorar a apresentação em vários lugares do livro. Se houvesse uma Calçada da Fama para editores técnicos, Julie seria, com certeza, a nossa candidata número um. Ela é nada menos que fenomenal. Obrigado, obrigado, obrigado, Julie! Priya Natarajan também descobriu alguns erros que pudemos corrigir antes da impressão deste livro. Quaisquer erros que restarem (e não temos dúvidas de que ainda há alguns) são da responsabilidade dos autores (e provavelmente foram inseridos depois que Julie leu o material).

O tratamento para as árvores de van Emde Boas baseia-se nas notas de Erik Demaine que, por sua vez, foram influenciadas por Michael Bender. Incorporamos também ideias de Javed Aslam, Bradley Kuszmaul e Hui Zha nesta edição.

O capítulo sobre multithreading foi baseado em notas escritas originalmente em conjunto com Harald Prokop. O material foi influenciado pelo de vários outros que trabalharam no projeto Cilk no MIT, entre eles Bradley Kuszmaul e Matteo Frigo. O projeto do pseudocódigo multithread foi inspirado pelas extensões Cilk do MIT para C e por extensões da Cilk++ para C++ pela Cilk Arts.

Agradecemos também aos muitos leitores da primeira e segunda edições que indicaram erros ou apresentaram sugestões para melhorar este livro. Corrigimos todos os erros de boa-fé indicados e incorporamos o máximo de sugestões possível. Ficamos contentes com o grande crescimento do número de tais contribuintes, mas tristes porque tornou-se impraticável nomear todos eles.

Finalmente, agradecemos a nossas esposas — Nicole Cormen, Wendy Leiserson, Gail Rivest e Rebecca Ivry — e a nossos filhos — Ricky, Will, Debby e Katie Leiserson; Alex e Christopher Rivest; e Molly, Noah e Benjamin Stein — por seu amor e apoio durante a preparação deste livro. A paciência e o estímulo de nossas famílias tornou esse projeto possível. Dedicamos este livro afetuosamente a elas.

THOMAS H. CORMEN — *Lebanon*, *New Hampshire*

CHARLES E. LEISERSON — *Cambridge*, *Massachusetts*

RONALD L. RIVEST — *Cambridge*, *Massachusetts*

CLIFFORD STEIN — *New York*, *New York*

Fevereiro de 2009

---

[1] Investigamos vários programas de desenho que são executados sob Mac OS X, mas todos apresentaram deficiências significativas em comparação com o MacDraw Pro. Por algum tempo tentamos produzir as ilustrações para este livro com um programa de desenho diferente, muito conhecido. Constatamos que demorava no mínimo cinco vezes mais tempo para produzir cada ilustração do que com o MacDraw Pro, e as ilustrações resultantes não eram tão boas. Daí a decisão de voltar para o MacDraw Pro executado em Macintoshes mais antigos.

SUMÁRIO

Parte

FUNDAMENTOS

INTRODUÇÃO

Esta parte o fará refletir sobre o projeto e a análise de algoritmos. Ela foi planejada para ser uma introdução suave ao modo como especificamos algoritmos, a algumas das estratégias de projeto que usaremos ao longo deste livro e a muitas das ideias fundamentais empregadas na análise de algoritmos. As partes posteriores deste livro serão elaboradas sobre essa base.

O Capítulo 1 é uma visão geral de algoritmos e de seu lugar em modernos sistemas de computação. Esse capítulo define o que é um algoritmo e dá uma lista com alguns exemplos. Além disso, traz a tese de que devemos considerar algoritmos como uma tecnologia, lado a lado com hardware rápido, interfaces gráficas do usuário, sistemas orientados a objetos e redes.

No Capítulo 2, veremos nossos primeiros algoritmos, que resolvem o problema de ordenar uma sequência de n números. Eles são escritos em um pseudocódigo que, embora não possa ser traduzido diretamente para nenhuma linguagem de programação convencional, transmite a estrutura do algoritmo com clareza suficiente para que você possa implementá-la na linguagem de sua preferência. Os algoritmos de ordenação que examinamos são a ordenação por inserção, que utiliza uma abordagem incremental, e a ordenação por intercalação, que usa uma técnica recursiva conhecida como "divisão e conquista". Embora o tempo exigido por esses dois algoritmos aumente com o valor n, a taxa de aumento de cada um é diferente. Determinamos esses tempos de execução no Capítulo 2 e desenvolvemos uma notação útil para expressá-los.

O Capítulo 3 define com exatidão essa notação, que denominamos notação assintótica. O capítulo começa definindo várias notações assintóticas que utilizamos para limitar os tempos de execução dos algoritmos por cima e por baixo. O restante do Capítulo 3 é primariamente uma apresentação da notação matemática, cuja finalidade é mais a de assegurar que o uso que você faz da notação corresponda à que fazemos neste livro do que lhe ensinar novos conceitos matemáticos.

O Capítulo 4 examina mais a fundo o método de divisão e conquista apresentado no Capítulo 2. Dá exemplos adicionais de algoritmos de divisão e conquista, incluindo o surpreendente método de Strassen para multiplicação de duas matrizes quadradas. O Capítulo 4 contém métodos para solução de recorrências que são úteis para descrever os tempos de execução de algoritmos recursivos. Uma técnica eficiente é o "método mestre", que frequentemente usamos para resolver recorrências que surgem dos algoritmos de divisão e conquista. Embora grande parte do Capítulo 4 seja dedicada a demonstrar a correção do método mestre, você pode saltar essa demonstração e ainda assim empregar o método mestre.

O Capítulo 5 introduz análise probabilística e algoritmos aleatorizados. Normalmente usamos análise probabilística para determinar o tempo de execução de um algoritmo em casos em que, devido à presença de uma distribuição de probabilidades inerente, o tempo de execução pode ser diferente para entradas diferentes do mesmo tamanho. Em alguns casos, consideramos que as entradas obedecem a uma distribuição de probabilidades conhecida, de modo que

calculamos o tempo de execução médio para todas as entradas possíveis. Em outros casos, a distribuição de probabilidades não vem das entradas, mas de escolhas aleatórias feitas durante o curso do algoritmo. Um algoritmo cujo comportamento seja determinado não apenas por sua entrada mas também pelos valores produzidos por um gerador de números aleatórios é um algoritmo aleatorizado. Podemos usar algoritmos aleatorizados para impor uma distribuição de probabilidade às entradas — garantindo assim que nenhuma entrada específica sempre cause fraco desempenho — ou mesmo para limitar a taxa de erros de algoritmos que têm permissão para produzir resultados incorretos em base limitada.

Os apêndices A-D contêm outro material matemático que você verá que são úteis à medida que ler este livro. É provável que você tenha visto grande parte do material dos apêndices antes de ler este livro (embora as definições e convenções específicas de notação que usamos possam ser diferentes em alguns casos daquelas que você já viu) e, portanto, você deve considerar os apêndices material de referência. Por outro lado, é provável que você ainda não tenha visto a maior parte do material contido na Parte I. Todos os capítulos da Parte I e os apêndices foram escritos com um toque de tutorial.

1 O PAPEL DOS ALGORITMOS NA COMPUTAÇÃO

O que são algoritmos? Por que o estudo dos algoritmos vale a pena? Qual é o papel dos algoritmos em relação a outras tecnologias usadas em computadores? Neste capítulo, responderemos a essas perguntas.

1.1 ALGORITMOS

Informalmente, um ***algoritmo*** é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como ***entrada*** e produz algum valor ou conjunto de valores como ***saída***. Portanto, um algoritmo é uma sequência de etapas computacionais que transformam a entrada na saída.

Também podemos considerar um algoritmo como uma ferramenta para resolver um ***problema computacional*** bem especificado. O enunciado do problema especifica em termos gerais a relação desejada entre entrada e saída. O algoritmo descreve um procedimento computacional específico para se conseguir essa relação entre entrada e saída.

Por exemplo, poderia ser necessário ordenar uma sequência de números em ordem não decrescente. Esse problema surge com frequência na prática e oferece um solo fértil para a apresentação de muitas técnicas de projeto e ferramentas de análise padronizadas. Vejamos como definir formalmente o ***problema de ordenação***:

Entrada: Uma sequência de n números $\langle a_1, a_2, ..., a_n \rangle$.

Saída: Uma permutação (reordenação) $\langle a_1', a_2', ..., a_n' \rangle$ da sequência de entrada, tal que $a_1' \leq a_2' \leq ... \leq a_n'$.

Por exemplo, dada a sequência de entrada $\langle 31, 41, 59, 26, 41, 58 \rangle$, um algoritmo de ordenação devolve como saída a sequência $\langle 26, 31, 41, 41, 58, 59 \rangle$. Tal sequência de entrada é denominada ***instância*** do problema de ordenação. Em geral, uma ***instância de um problema*** consiste na entrada (que satisfaz quaisquer restrições impostas no enunciado do problema) necessária para calcular uma solução para o problema.

Como muitos programas a utilizam como etapa intermediária, a ordenação é uma operação fundamental em ciência da computação e, por isso, há um grande número de bons algoritmos de ordenação à nossa disposição. O melhor algoritmo para determinada aplicação depende — entre outros fatores — do número de itens a ordenar, do grau de ordenação já apresentado por esses itens, das possíveis restrições aos valores dos itens, da arquitetura do computador e do tipo de dispositivo de armazenamento que será utilizado: memória principal, discos ou até mesmo fitas.

Diz-se que um algoritmo é ***correto*** se, para toda instância de entrada, ele parar com a saída correta. Dizemos que um algoritmo correto ***resolve*** o problema computacional dado. Um algoritmo incorreto poderia não parar em algumas instâncias de entrada ou poderia parar com uma resposta incorreta. Ao contrário do que se poderia esperar, às vezes os algoritmos incorretos podem ser úteis, se pudermos controlar sua taxa de erros. No Capítulo 31, veremos um exemplo de algoritmo com taxa de erro controlável quando estudarmos algoritmos para encontrar grandes números primos. Porém, de modo geral, nos concentraremos apenas em algoritmos corretos.

Um algoritmo pode ser especificado em linguagem comum como um programa de computador ou mesmo como um projeto de hardware. O único requisito é que a especificação deve fornecer uma descrição precisa do procedimento computacional a ser seguido.

Que tipos de problemas são resolvidos por algoritmos?

A ordenação não é de modo algum o único problema computacional para o qual os algoritmos foram desenvolvidos (e é provável que você já tenha suspeitado disso quando viu o tamanho deste livro.). As aplicações práticas de algoritmos estão por toda parte e incluem os exemplos a seguir:

- O Projeto Genoma Humano vem alcançando grande progresso no cumprimento de suas metas de identificar todos os 100.000 genes do DNA humano, determinar as sequências dos três bilhões de pares de bases químicas que constituem o DNA humano, armazenar essas informações em bancos de dados e desenvolver ferramentas para análise de dados. Cada uma dessas etapas exige algoritmos sofisticados. Embora as soluções para os vários problemas envolvidos estejam fora do escopo deste livro, muitos métodos aplicados à resolução desses problemas biológicos usam ideias apresentadas em vários capítulos deste livro, permitindo que os cientistas realizem tarefas e, ao mesmo tempo, utilizem os recursos com eficiência. As economias são de tempo, tanto humano quanto de máquina, e de dinheiro, já que mais informações podem ser extraídas de técnicas de laboratório.
- A Internet permite que pessoas em todo o mundo acessem e obtenham rapidamente grande quantidade de informações. Com o auxílio de algoritmos engenhosos, sites da Internet conseguem gerenciar e manipular esse grande volume de dados. Entre os exemplos de problemas que dependem essencialmente da utilização de algoritmos citamos a determinação de boas rotas para a transmissão de dados (técnicas para resolver tais problemas são apresentadas no Capítulo 24) e a utilização de um mecanismo de busca para encontrar rapidamente páginas em que estão determinadas informações (técnicas relacionadas são apresentadas nos Capítulos 11 e 32).
- O comércio eletrônico permite que mercadorias e serviços sejam negociados e trocados eletronicamente e depende do sigilo de informações pessoais, como números de cartões de crédito, senhas e extratos bancários. Entre as principais tecnologias utilizadas no comércio eletrônico estão a criptografia de chave pública e as assinaturas digitais (estudadas no Capítulo 31), ambas baseadas em algoritmos numéricos e na teoria dos números.
- Na indústria e em outros empreendimentos comerciais, muitas vezes é preciso alocar recursos escassos da maneira mais benéfica possível. Uma empresa petrolífera talvez deseje saber onde localizar seus poços para maximizar o lucro esperado. Um político talvez queira determinar onde gastar dinheiro em publicidade de campanha para maximizar as chances de vencer uma eleição. Uma empresa de transporte aéreo poderia querer designar tripulações a voos da forma menos dispendiosa possível, garantindo que cada voo seja atendido e que as regulamentações do governo relativas à escala das tripulações sejam obedecidas. Um provedor de serviços da Internet talvez queira definir onde alocar recursos adicionais para servir a seus clientes com mais eficiência. Todos esses são exemplos de problemas que podem ser resolvidos com a utilização de programação linear, que estudaremos no Capítulo 29.

Embora alguns dos detalhes desses exemplos estejam fora do escopo deste livro, forneceremos técnicas básicas que se aplicam a esses problemas e áreas de problemas. Também mostraremos como resolver muitos problemas específicos, inclusive os seguintes:

- Temos um mapa rodoviário no qual estão marcadas as distâncias entre cada par de interseções adjacentes e queremos determinar a rota mais curta entre uma interseção e outra. O número de rotas possíveis pode ser enorme, ainda que sejam descartadas as rotas que se entrecruzam. Como escolher a mais curta de todas as rotas possíveis? Aqui, modelamos o mapa rodoviário (que é ele próprio um modelo das estradas reais) como um grafo (o que veremos na Parte VI e no Apêndice B) e desejamos determinar o caminho mais curto de um vértice até outro no grafo. Veremos como resolver esse problema com eficiência no Capítulo 24.
- Temos duas sequências ordenadas de símbolos, $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2,..., y_n \rangle$, e queremos determinar uma subsequência comum mais longa de X e Y. Uma subsequência de X é apenas X com alguns (ou, possivelmente, todos ou nenhum) de seus elementos removidos. Por exemplo, uma subsequência de $\langle A, B, C, D, E, F, G \rangle$ seria $\langle B, C, E, G \rangle$. O comprimento de uma subsequência comum mais longa de X e Y nos dá uma ideia do grau de semelhança dessas duas sequências. Por exemplo, se as duas sequências forem pares de base em

filamentos de DNA, poderemos considerá-las semelhantes se tiverem uma subsequência comum longa. Se X tiver m símbolos e Y tiver n símbolos, então X e Y terão 2^m e 2_n possíveis subsequências, respectivamente. Selecionar todas as possíveis subsequências de X e Y e então combiná-las poderá tomar um tempo proibitivamente longo, a menos que m e n sejam muito pequenos. No Capítulo 15 veremos como usar uma técnica geral conhecida como programação dinâmica para resolver esse problema com eficiência muito maior.

- Temos um projeto mecânico apresentado como um catálogo de peças no qual cada uma pode incluir instâncias de outras peças e precisamos organizar uma lista ordenada de peças de modo que cada uma apareça antes de qualquer peça que a utilize. Se o projeto compreender n peças, então haverá $n!$ ordenações possíveis, onde $n!$ denota a função fatorial. Como a função fatorial cresce ainda mais rapidamente do que uma função exponential, não existe uma possibilidade viável de gerar cada ordenação possível e então verificar se, dentro daquela ordenação, cada peça aparece antes das peças que a utilizam (a menos que tenhamos apenas um pequeno número de peças). Esse problema é uma instância de ordenação topológica, e estudaremos como resolvê-lo com eficiência no Capítulo 22.
- Temos n pontos no plano e desejamos determinar a envoltória convexa desses pontos. A envoltória convexa é o menor polígono convexo que contém os pontos. Intuitivamente, podemos imaginar que cada ponto seja representado pela cabeça saliente de um prego fixado a uma tábua. A envoltória convexa seria representada por um elástico apertado que contorna todos os pregos. Cada prego pelo qual o elástico passar é um vértice da envoltória convexa (veja um exemplo na Figura 33.6). Qualquer um dos 2_n subconjuntos dos pontos poderia ser os vértices da envoltória convexa. Porém, saber quais pontos são vértices da envoltória convexa não é suficiente, já que também precisamos conhecer a ordem em que eles aparecem. Portanto, há muitas escolhas para os vértices da envoltória convexa. O Capítulo 33 apresenta dois bons métodos para determinar a envoltória convexa.

Essas listas estão longe de esgotar os exemplos (como é provável que você já tenha imaginado de novo pelo peso deste livro), mas exibem duas características comuns a muitos problemas algorítmicos interessantes:

1. Eles têm muitas soluções candidatas, a grande maioria das quais não resolve o problema que temos em mãos. Encontrar uma solução que o resolva, ou uma solução que seja a "melhor", pode representar um desafio significativo.
2. Eles têm aplicações práticas. Dentre os problemas da lista que apresentamos, o da determinação do caminho mais curto é o que fornece os exemplos mais fáceis. Uma empresa de transporte rodoviário ou ferroviário tem interesse financeiro em determinar os caminhos mais curtos em uma rede rodoviária ou ferroviária porque percursos menores resultam em menores custos de mão de obra e combustível. Ou um nó de roteamento na Internet pode precisar encontrar o caminho mais curto através da rede, para rotear uma mensagem com rapidez. Ou pode ser que alguém que deseje viajar de carro de Nova York a Boston queira encontrar instruções em um site Web adequado ou usar seu GPS durante o percurso.

Nem todo problema resolvido por algoritmos tem um conjunto de soluções candidatas fáceis de identificar. Por exemplo, suponha que temos um conjunto de valores numéricos que representam amostras de um sinal e que queremos calcular a transformada discreta de Fourier dessas amostras. A transformada discreta de Fourier converte o domínio do tempo para o domínio da frequência, produzindo um conjunto de coeficientes numéricos, de modo que podemos determinar a força de várias frequências no sinal amostrado. Além de estarem no cerne do processamento de sinais, as transformadas discretas de Fourier também se aplicam à compressão de dados e à multiplicação de grandes polinômios e inteiros. O Capítulo 30 apresenta um algoritmo eficiente para esse problema, a transformada rápida de Fourier (comumente denominada FFT), e também apresenta o esquema de projeto de um circuito de *hardware* para calcular a FFT.

Estruturas de dados

Este livro também contém várias estruturas de dados. Uma ***estrutura de dados*** é um modo de armazenar e organizar dados com o objetivo de facilitar acesso e modificações. Nenhuma estrutura de dados única funciona bem para todas as finalidades e, por isso, é importante conhecer os pontos fortes e as limitações de várias delas.

Técnica

Embora você possa usar este livro como um "livro de receitas" para algoritmos, é possível que algum dia encontre um problema cujo algoritmo publicado (muitos dos exercícios e problemas deste livro, por exemplo) não consiga achar imediatiamente. Este livro lhe ensinará técnicas de projeto e análise de algoritmos para que você possa desenvolver algoritmos por conta própria, mostrar que eles fornecem a resposta correta e entender sua eficiência.

Capítulos diferentes abordam aspectos diferentes da solução de problemas de algoritmos. Alguns abordam problemas específicos, como determinar medianas e ordenar dados estatísticos, no Capítulo 9, calcular árvores geradoras (*spanning trees*) mínimas, no Capítulo 23, e determinar um fluxo máximo em uma rede, no Capítulo 26. Outros capítulos abordam técnicas como a de divisão e conquista, no Capítulo 4, programação dinâmica, no Capítulo 15, e análise amortizada, no Capítulo 17.

Problemas difíceis

A maior parte deste livro trata de algoritmos eficientes. Nossa medida habitual de eficiência é a velocidade, isto é, quanto tempo um algoritmo demora para produzir seu resultado. Porém, existem alguns problemas para os quais não se conhece nenhuma solução eficiente. O Capítulo 34 estuda um subconjunto interessante desses problemas, conhecidos como NP-completos.

Por que os problemas NP-completos são interessantes? Em primeiro lugar, embora nenhum algoritmo eficiente para um problema NP-completo tenha sido encontrado até agora, ninguém jamais provou que não é possível existir um algoritmo eficiente para tal problema. Em outras palavras, ninguém sabe se existem ou não algoritmos eficientes para problemas NP-completos. Em segundo lugar, o conjunto de problemas NP-completos tem a propriedade notável de que, se existir um algoritmo eficiente para qualquer um deles, então existem algoritmos eficientes para todos eles. Essa relação entre os problemas NP-completos torna a falta de soluções eficientes ainda mais torturante. Em terceiro lugar, vários problemas NP-completos são semelhantes mas não idênticos a problemas para os quais sabemos existir algoritmos eficientes. Cientistas da computação ficam intrigados com o fato de que uma pequena mudança no enunciado do problema pode provocar uma grande alteração na eficiência do melhor algoritmo conhecido.

É bom que você conheça os problemas NP-completos porque alguns deles surgem com frequência surpreendente em aplicações reais. Se você tiver de produzir um algoritmo eficiente para um problema NP-completo, é provável que perca muito tempo em uma busca infrutífera. Por outro lado, se você conseguir mostrar que o problema é NP-completo, poderá dedicar seu tempo ao desenvolvimento de um algoritmo eficiente que ofereça uma solução boa, embora não a melhor possível.

Como exemplo concreto, considere uma empresa transportadora que tenha um depósito central. Todo dia, cada caminhão é carregado no depósito e enviado a diversos locais para fazer entregas. No final do dia, cada caminhão tem de estar de volta ao depósito para ser preparado para a carga no dia seguinte. Para reduzir custos, a empresa quer selecionar uma ordem de pontos de entrega que represente a menor distância total a ser percorrida por cada caminhão. Esse problema é o famoso "problema do caixeiro-viajante", e é NP-completo — não tem nenhum algoritmo eficiente conhecido. Contudo, adotando-se certas premissas, há algoritmos eficientes que fornecem uma distância total não muito acima da menor possível. O Capítulo 35 discute esses "algoritmos de aproximação".

Paralelismo

Durante muitos anos pudemos contar com uma taxa regular de aumento da velocidade de relógio dos processadores. Porém, limitações físicas representam um entrave fundamental ao crescimento constante dessas

velocidades: como a densidade de potência aumenta superlinearmente com a velocidade de relógio, existe o risco de derretimento dos chips quando essas velocidades atingem um certo nível. Portanto, para executar mais cálculos por segundo, o projeto moderno de chips prevê não apenas um, mas vários "núcleos" de processamento. Podemos comparar esses computadores com vários núcleos a vários computadores sequenciais em um único chip; em outras palavras, eles são um tipo de "computador paralelo". Para obter o melhor desempenho desses processadores com vários núcleos, precisamos produzir algoritmos que considerem o paralelismo. O Capítulo 27 apresenta um modelo de algoritmo "multithread" que tira proveito de núcleos múltiplos. Do ponto de vista teórico, esse modelo é vantajoso e constitui a base de vários modelos eficientes de programas de computador, entre eles um programa para campeonatos de xadrez.

Exercícios

1.1-1 Cite um exemplo real que exija ordenação ou um exemplo real que exija o cálculo de uma envoltória convexa.

1.1-2 Além da velocidade, que outras medidas de eficiência poderiam ser usadas em uma configuração real?

1.1-3 Selecione uma estrutura de dados que você já tenha visto antes e discuta seus pontos fortes e suas limitações.

1.1-4 Em que aspectos os problemas anteriores do caminho mais curto e do caixeiro-viajante são semelhantes? Em que aspectos eles são diferentes?

1.1-5 Mostre um problema real no qual apenas a melhor solução servirá. Em seguida, apresente um problema em que baste uma solução que seja "aproximadamente" a melhor.

1.2 ALGORITMOS COMO TECNOLOGIA

Suponha que os computadores fossem infinitamente rápidos e que a memória do computador fosse gratuita. Você teria alguma razão para estudar algoritmos? A resposta é sim, ainda que apenas porque você gostaria de demonstrar que seu método de solução termina, e o faz com a resposta correta.

Se os computadores fossem infinitamente rápidos, qualquer método correto para resolver um problema serviria. É provável que você quisesse que sua implementação estivesse dentro dos limites da boa prática de engenharia de software (isto é, que ela fosse bem documentada e projetada) mas, na maior parte das vezes, você utilizaria o método que fosse mais fácil de implementar.

É claro que os computadores podem ser rápidos, mas eles não são infinitamente rápidos. A memória pode ser de baixo custo, mas não é gratuita. Assim, o tempo de computação é um recurso limitado, bem como o espaço na memória. Esses recursos devem ser usados com sensatez, e algoritmos eficientes em termos de tempo ou espaço o ajudarão a usá-los assim.

Eficiência

Algoritmos diferentes criados para resolver o mesmo problema muitas vezes são muito diferentes em termos de eficiência. Essas diferenças podem ser muito mais significativas que as diferenças relativas a hardware e software.

Como exemplo, no Capítulo 2 estudaremos dois algoritmos de ordenação. O primeiro, conhecido como ***ordenação por inserção***, leva um tempo aproximadamente igual a $c_1 n^2$ para ordenar n itens, onde c_1 é uma constante que não depende de n. Isto é, ele demora um tempo aproximadamente proporcional a n_2. O segundo, de ***ordenação por intercalação***, leva um tempo aproximadamente igual a $c_2 n \lg n$, onde $\lg n$ representa $\log^2 n$ e c_2 é outra constante que também não depende de n. A ordenação por inserção normalmente tem um fator constante menor que a ordenação

por intercalação; assim, $c_1 < c_2$. Veremos que os fatores constantes podem causar um impacto muito menor sobre o tempo de execução que a dependência do tamanho da entrada n. Se representarmos o tempo de execução da ordenação por inserção por $c_1 n \cdot n$ e o tempo de execução da ordenação por intercalação por $c_2 n \cdot \lg n$, veremos que, enquanto o fator do tempo de execução da ordenação por inserção é n, o da ordenação por intercalação é $\lg n$, que é muito menor (por exemplo, quando $n = 1.000$, $\lg n$ é aproximadamente 10 e quando n é igual a um milhão, $\lg n$ é, aproximadamente, só 20). Embora a ordenação por inserção em geral seja mais rápida que a ordenação por intercalação para pequenos tamanhos de entradas, tão logo o tamanho da entrada n se torne grande o suficiente a vantagem da ordenação por intercalação de $\lg n$ contra n compensará com sobras a diferença em fatores constantes. Independentemente do quanto c_1 seja menor que c_2, sempre haverá um ponto de corte além do qual a ordenação por intercalação será mais rápida.

Como exemplo concreto, vamos comparar um computador mais rápido (computador A) que executa a ordenação por inserção com um computador mais lento (computador B) que executa a ordenação por intercalação. Cada um deve ordenar um arranjo de dez milhões de números. (Embora 10 milhões de números possa parecer muito, se os números forem inteiros de oito bytes, a entrada ocupará cerca de 80 megabytes e caberá com grande folga até mesmo na memória de um laptop barato.) Suponha que o computador A execute dez bilhões de instruções por segundo (mais rapidamente do que qualquer computador sequencial existente na época da redação deste livro) e que o computador B execute apenas dez milhões de instruções por segundo; assim, o computador A será 1.000 vezes mais rápido que o computador B em capacidade bruta de computação. Para tornar a diferença ainda mais drástica, suponha que o programador mais astucioso do mundo codifique a ordenação por inserção em linguagem de máquina para o computador A e que o código resultante exija $2n_2$ instruçõespara ordenar n números. Suponha ainda que um programador médio implemente a ordenação por intercalação utilizando uma linguagem de alto nível com um compilador ineficiente, sendo que o código resultante totaliza $50n \lg n$ instruções. Para ordenar 10 milhões de números, o computador A leva

$$\frac{2 \cdot (10^7)^2 \text{ instruções}}{10^{10} \text{ instruções/segundo}} = 20.000 \text{ segundos (mais de 5,5 horas);}$$

por outro lado, o computador B leva

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 1163 \text{ segundos (menos de 20 minutos).}$$

Usando um algoritmo cujo tempo de execução cresce mais lentamente, até mesmo com um compilador fraco, o computador B funciona mais de 17 vezes mais rapidamente que o computador A! A vantagem da ordenação por intercalação é ainda mais evidente quando ordenamos 100 milhões de números: onde a ordenação por inserção demora mais de 23 dias, a ordenação por intercalação demora menos de quatro horas. Em geral, à medida que o tamanho do problema aumenta, também aumenta a vantagem relativa da ordenação por intercalação.

Algoritmos e outras tecnologias

O exemplo anterior mostra que os algoritmos, como o hardware de computadores, devem ser considerados como uma ***tecnologia***. O desempenho total do sistema depende da escolha de algoritmos eficientes tanto quanto da escolha de hardware rápido. Os rápidos avanços que estão ocorrendo em outras tecnologias computacionais também estão sendo observados em algoritmos. Você poderia questionar se os algoritmos são verdadeiramente tão importantes para os computadores contemporâneos levando em consideração outras tecnologias avançadas, como:

- arquiteturas computacionais e tecnologias de fabricação avançadas;
- interfaces gráficas de usuário (GUIs) intuitivas e fáceis de usar;
- sistemas orientados a objetos;

- tecnologias integradas da Web e
- redes de alta velocidade, com fio e sem fio.

A resposta é: sim. Embora algumas aplicações não exijam explicitamente conteúdo algorítmico no nível da aplicação (como algumas aplicações simples baseadas na Web), a maioria exige. Por exemplo, considere um serviço da Web que determina como viajar de um local para outro. Sua implementação dependeria de hardware rápido, de uma interface gráfica de usuário, de redes remotas, além de, possivelmente, orientação a objetos. Contudo, também exigiria algoritmos para certas operações, como descobrir rotas (talvez empregando um algoritmo de caminho mais curto), apresentar mapas e interpolar endereços.

Além disso, até mesmo uma aplicação que não exija conteúdo algorítmico no nível da aplicação depende muito de algoritmos. A aplicação depende de hardware rápido? O projeto de hardware utilizou algoritmos. A aplicação depende de interfaces gráficas de usuário? O projeto de qualquer GUI depende de algoritmos. A aplicação depende de rede? O roteamento em redes depende muito de algoritmos. A aplicação foi escrita em uma linguagem diferente do código de máquina? Então, ela foi processada por um compilador, um interpretador ou um montador, e todos fazem uso extensivo de algoritmos. Os algoritmos estão no núcleo da maioria das tecnologias usadas em computadores contemporâneos.

Além disso, com a capacidade cada vez maior dos computadores, nós os utilizamos mais do que nunca para resolver problemas cada vez maiores. Como vimos na comparação anterior entre ordenação por inserção e ordenação por intercalação, é nos problemas maiores que as diferenças entre a eficiência dos algoritmos se tornam particularmente notáveis.

Uma sólida base de conhecimento e técnica de algoritmos é uma das características que separam os programadores verdadeiramente qualificados dos novatos. Com a moderna tecnologia de computação, você pode executar algumas tarefas sem saber muito sobre algoritmos; porém, com uma boa base em algoritmos, é possível fazer muito, muito mais.

Exercícios

1.2-1 Cite um exemplo de aplicação que exige conteúdo algorítmico no nível da aplicação e discuta a função dos algoritmos envolvidos.

1.2-2 Suponha que estamos comparando implementações de ordenação por inserção e ordenação por intercalação na mesma máquina. Para entradas de tamanho n, a ordenação por inserção é executada em $8n_2$ passos, enquanto a ordenação por intercalação é executada em $64n \lg n$ passos. Para quais valores de n a ordenação por inserção supera a ordenação por intercalação?

1.2-3 Qual é o menor valor de n tal que um algoritmo cujo tempo de execução é $100n_2$ funciona mais rapidamente que um algoritmo cujo tempo de execução é 2_n na mesma máquina?

Problemas

1-1 Comparação entre tempos de execução

Para cada função $f(n)$ e cada tempo t na tabela a seguir, determine o maior tamanho n de um problema que pode ser resolvido no tempo t, considerando que o algoritmo para resolver o problema demore $f(n)$ microssegundos.

	1 segundo	1 minuto	1 hora	1 dia	1 mês	1 ano	1 século
$\lg n$							

$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

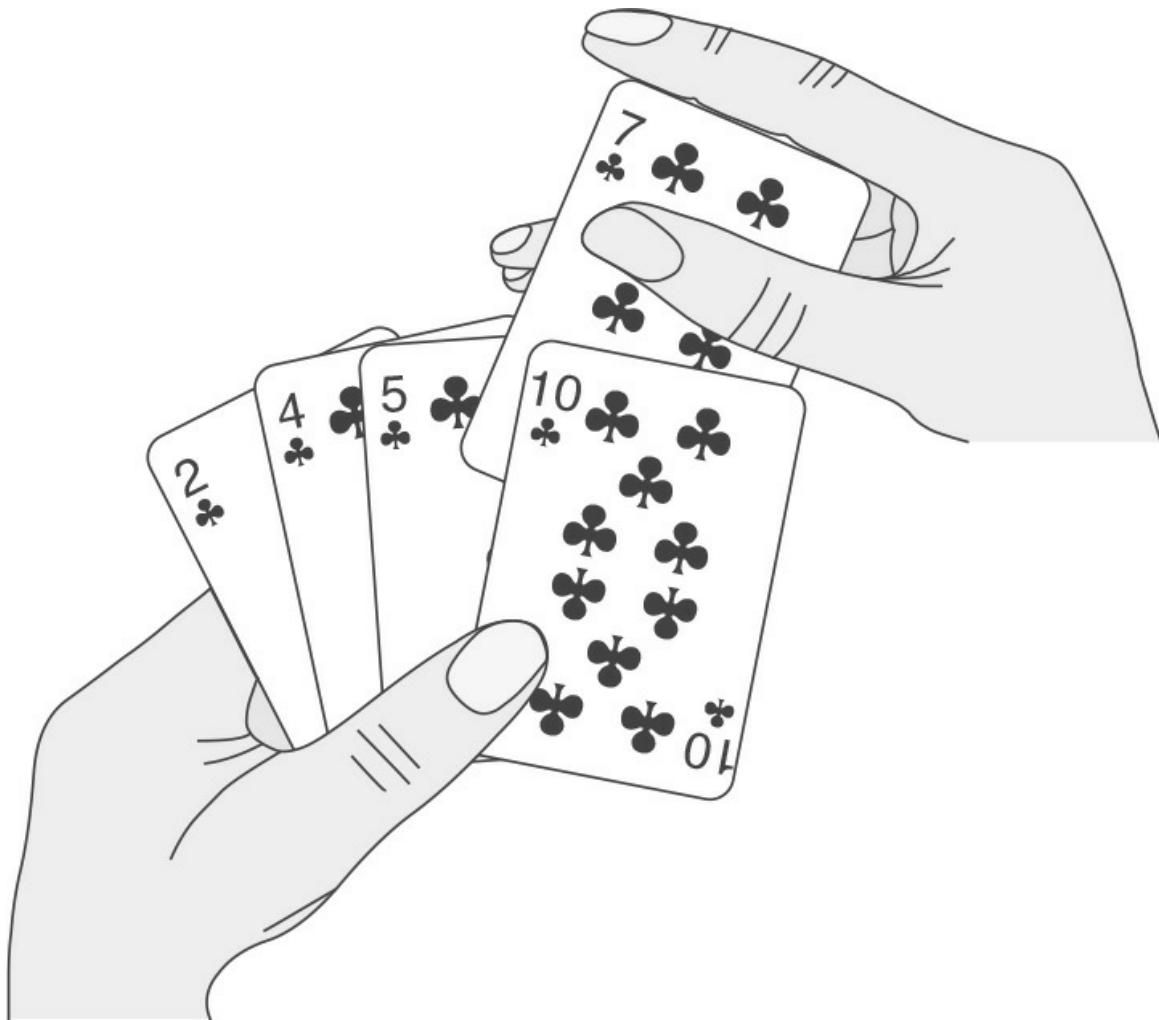
NOTAS DO CAPÍTULO

Existem muitos textos excelentes sobre o tópico geral de algoritmos, entre eles os de Aho, Hopcroft e Ullman [5, 6]; Baase e Van Gelder [28]; Brassard e Bratley [54]; Dasgupta, Papadimitriou e Vazirani [82]; Goodrich e Tamassia [148]; Hofri [175]; Horowitz, Sahni e Rajasekaran [181]; Johnsonbaugh e Schaefer [193]; Kingston [205]; Kleinberg e Tardos [208]; Knuth [209, 210, 211]; Kozen [220]; Levitin [235]; Manber [242]; Mehlhorn [249, 250, 251]; Purdom e Brown [287]; Reingold, Nievergelt e Deo [293]; Sedgewick [306]; Sedgewick e Flajolet [307]; Skiena [318] e Wilf [356]. Alguns dos aspectos mais práticos do projeto de algoritmos são discutidos por Bentley [42, 43] e Gonnet [145]. Pesquisas na área de algoritmos também podem ser encontradas no *Handbook of Theoretical Computer Science, Volume A* [342] e no CRC *Algorithms and Theory of Computation Handbook* [25]. Avaliações de algoritmos usados em biologia computacional podem ser encontrados nos livros didáticos de Gusfield [156], Pevzner [275], Setubal e Meidanis [310] e Waterman [350].

2 DANDO A PARTIDA

Este capítulo tem o objetivo de familiarizá-lo com a estrutura que usaremos em todo o livro para refletir sobre o projeto e a análise de algoritmos. Ele é autônomo, mas inclui diversas referências ao material que será apresentado nos Capítulos 3 e 4 (e também contém diversos somatórios, que o Apêndice A mostra como resolver).

Começaremos examinando o algoritmo de ordenação por inserção para resolver o problema de ordenação apresentado no Capítulo 1. Definiremos um "pseudocódigo" que deverá ser familiar aos leitores que tenham estudado programação de computadores, e o empregaremos com a finalidade de mostrar como serão especificados nossos algoritmos. Tendo especificado o algoritmo de ordenação por inserção, demonstraremos que ele efetua a ordenação corretamente e analisaremos seu tempo de execução. A análise introduzirá uma notação que focaliza o modo como o tempo aumenta com o número de itens a ordenar. Seguindo nossa discussão da ordenação por inserção, introduziremos a abordagem de divisão e conquista para o projeto de algoritmos e a utilizaremos para desenvolver um algoritmo denominado ordenação por intercalação. Terminaremos com uma análise do tempo de execução da ordenação por intercalação.

2.1 ORDENAÇÃO POR INSERÇÃO

Nosso primeiro algoritmo, o de ordenação por inserção, resolve o ***problema de ordenação*** apresentado no Capítulo 1:

Entrada: Uma sequência de n números $\langle a_1, a_2, ..., a_n \rangle$.

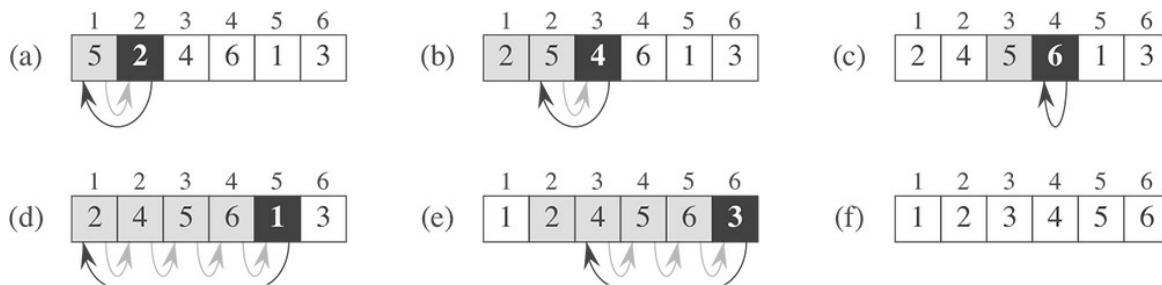
Saída: Uma permutação (reordenação) $\langle a'_1, a'_2,..., a'_n \rangle$ da sequência de entrada, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Os números que desejamos ordenar também são conhecidos como ***chaves***. Embora conceitualmente estejamos ordenando uma sequência, a entrada é dada na forma de um arranjo com n elementos.

Neste livro, descreveremos tipicamente algoritmos como programas escritos em um ***pseudocódigo*** semelhante em vários aspectos a C, C++, Java, Python ou Pascal. Se você já conhece qualquer dessas linguagens, deverá ter pouca dificuldade para ler nossos algoritmos. O que distingue o pseudocódigo do código "real" é que, no pseudocódigo, empregamos qualquer método expressivo que seja mais claro e conciso para especificar um dado algoritmo. Às vezes, o método mais claro é a linguagem comum; assim, não se surpreenda se encontrar uma frase ou sentença em nosso idioma (ou em inglês) embutida em uma seção de código "real". Outra diferença entre o pseudocódigo e o código real é que o pseudocódigo em geral não se preocupa com questões de engenharia de software. As questões de abstração de dados, modularidade e tratamento de erros são frequentemente ignoradas, de modo a transmitir a essência do algoritmo de modo mais conciso.

Figura 2.1 Ordenando cartas com o uso da ordenação por inserção.

Começaremos com a ***ordenação por inserção***, um algoritmo eficiente para ordenar um número pequeno de elementos. A ordenação por inserção funciona da maneira como muitas pessoas ordenam as cartas em um jogo de baralho. Iniciamos com a mão esquerda vazia e as cartas viradas para baixo, na mesa. Em seguida, retiramos uma carta de cada vez da mesa e a inserimos na posição correta na mão esquerda. Para encontrar a posição correta para uma carta, nós a comparamos com cada uma das cartas que já estão na mão, da direita para a esquerda, como ilustra a Figura 2.1. Em todas as vezes, as cartas que seguramos na mão esquerda são ordenadas, e essas cartas eram as que estavam na parte superior da pilha sobre a mesa.

Nosso pseudocódigo para ordenação por inserção é apresentado como um procedimento denominado Insertion-Sort, que toma como parâmetro um arranjo $A[1\,..\,n]$ contendo uma sequência de comprimento n que deverá ser ordenada. (No código, o número n de elementos em A é denotado por $A \cdot \mathit{comprimento}$.) O algoritmo ordena os números da entrada **no lugar:** reorganiza os números dentro do arranjo A, com no máximo um número constante deles armazenado fora do arranjo em qualquer instante. O arranjo de entrada A conterá a sequência de saída ordenada quando Insertion-Sort terminar.

```
INSERTION-SORT(A)
1  for j = 2 to A·comprimento
2      chave = A[j]
3      // Inserir A[j] na sequência ordenada A[1.. j − 1].
4      i = j − 1
5      while i > 0 e A[i] > chave
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = chave
```

Invariantes de laço e a correção da ordenação por inserção

A Figura 2.2 mostra como esse algoritmo funciona para $A = \langle 5, 2, 4, 6, 1, 3\rangle$. O índice j indica a "carta atual" que está sendo inserida na mão. No início de cada iteração do laço **for**, indexado por j, o subarranjo que consiste nos elementos $A[1 \,..\, j - 1]$ constitui a mão ordenada atualmente e o subconjunto remanescente $A[j + 1 \,..\, n]$ corresponde à pilha de cartas que ainda está sobre a mesa. Na verdade, os elementos $A[1 \,..\, j - 1]$ são os que estavam *originalmente* nas posições 1 a $j - 1$, mas agora em sequência ordenada. Afirmamos essas propriedades de $A[1 \,..\, j - 1]$ formalmente como um de *invariante de laço*:

Figura 2.2 A operação de INSERTION-SORT sobre o arranjo $A = \langle 5, 2, 4, 6, 1, 3\rangle$. Os índices do arranjo aparecem acima dos retângulos, e os valores armazenados nas posições do arranjo aparecem dentro dos retângulos. **(a)–(e)** Iterações do laço **for** das linhas 1 a 8. Em cada iteração, o retângulo preto contém a chave obtida de $A[j]$, que é comparada com os valores contidos nos retângulos sombreados à sua esquerda, no teste da linha 5. Setas sombreadas mostram os valores do arranjo deslocados uma posição para a direita na linha 6, e setas pretas indicam para onde a chave é deslocada na linha 8. **(f)** O arranjo ordenado final.

No início de cada iteração para o laço **for** das linhas 1–8, o subarranjo $A[1 \,..\, j - 1]$ consiste nos elementos que estavam originalmente em $A[1 \,..\, j - 1]$, porém em sequência ordenada.

Usamos invariantes de laço para nos ajudar a entender por que um algoritmo é correto. Devemos mostrar três detalhes sobre um invariante de laço:

Inicialização: Ele é verdadeiro antes da primeira iteração do laço.

Manutenção: Se ele for verdadeiro antes de uma iteração do laço, permanecerá verdadeiro antes da próxima iteração.

Término: Quando o laço termina, o invariante nos fornece uma propriedade útil que ajuda a mostrar que o algoritmo é correto.

Quando as duas primeiras propriedades são válidas, o invariante de laço é verdadeiro antes de toda iteração do laço. (É claro que temos a liberdade de usar fatos confirmados além do invariante de laço em si para provar que ele permanece verdadeiro antes de cada iteração.) Observe a semelhança com a indução; nesta, para provar que uma

propriedade é válida, provamos uma base e um passo de indução. Aqui, mostrar que o invariante é válido antes da primeira iteração equivale à base, e mostrar que o invariante é válido de uma iteração para outra equivale ao passo.

A terceira propriedade talvez seja a mais importante, visto que estamos usando o invariante de laço para mostrar a correção. Normalmente, usamos o invariante de laço juntamente com a condição que provocou o término do laço. O modo de utilização da propriedade de término é diferente do modo de utilização da indução: nesta, a etapa indutiva é aplicada indefinidamente; aqui, paramos a "indução" quando o laço termina.

Vamos ver como essas propriedades são válidas para ordenação por inserção:

Inicialização: Começamos mostrando que o invariante de laço é válido antes da primeira iteração do laço, quando $j = 2$.[1] Então, o subarranjo $A[1 .. j - 1]$ consiste apenas no único elemento $A[1]$, que é de fato o elemento original em $A[1]$. Além disso, esse subarranjo é ordenado (trivialmente, é claro), e isso mostra que o invariante de laço é válido antes da primeira iteração do laço.

Manutenção: Em seguida, abordamos a segunda propriedade: mostrar que cada iteração mantém o invariante de laço. Informalmente, o corpo do laço **for** funciona deslocando $A[j - 1]$, $A[j - 2]$, $A[j - 3]$, e assim por diante, uma posição para a direita até encontrar a posição adequada para $A[j]$ (linhas 4 a 7); nesse ponto ele insere o valor de $A[j]$ (linha 8). Então, o subarranjo $A[1 .. j]$ consiste nos elementos presentes originalmente em $A[1 .. j]$, mas em sequência ordenada. Portanto, incrementar j para a próxima iteração do laço **for** preserva o invariante de laço.

Um tratamento mais formal da segunda propriedade nos obrigaria a estabelecer e mostrar um invariante para o laço **while** das linhas 5–7. Porém, nesse momento, preferimos não nos prender a tal formalismo, e assim contamos com nossa análise informal para mostrar que a segunda propriedade é válida para o laço externo.

Término: Finalmente, examinamos o que ocorre quando o laço termina. A condição que provoca o término do laço **for** é que $j > A \cdot comprimento = n$. Como cada iteração do laço aumenta j de 1, devemos ter $j = n + 1$ nesse instante. Substituindo j por $n + 1$ no enunciado do invariante de laço, temos que o subarranjo $A[1 .. n]$ consiste nos elementos originalmente contidos em $A[1 .. n]$, mas em sequência ordenada. Observando que o subarranjo $A[1 .. n]$ é o arranjo inteiro, concluímos que o arranjo inteiro está ordenado. Portanto o algoritmo está correto.

Empregaremos esse método de invariantes de laço para mostrar a correção mais adiante neste capítulo e também em outros capítulos.

Convenções de pseudocódigo

Utilizaremos as convenções a seguir em nosso pseudocódigo.

- O recuo indica estrutura de bloco. Por exemplo, o corpo do laço for que começa na linha 1 consiste nas linhas 2 a 8, e o corpo do laço **while** que começa na linha 5 contém as linhas 6 e 7, mas não a linha 8. Nosso estilo de recuo também se aplica a instruções **if-else**.[2] O uso de recuo em vez de indicadores convencionais de estrutura de bloco, como instruções **begin** e **end**, reduz bastante a confusão, ao mesmo tempo que preserva ou até mesmo aumenta a clareza.[3]
- As interpretações das construções de laço **while**, **for** e **repeat-until** e das construções condicionais **if-else** são semelhantes às das linguagens C, C++, Java, Python e Pascal.[4] Neste livro, o contador do laço mantém seu valor após sair do laço, ao contrário de algumas situações que surgem em C++, Java e Pascal. Desse modo, logo depois de um laço **for**, o valor do contador de laço é o valor que primeiro excedeu o limite do laço **for**. Usamos essa propriedade em nosso argumento de correção para a ordenação por inserção. O cabeçalho do laço **for** na linha 1 é **for** $j = 2$ **to** $A \cdot comprimento$ e, assim, quando esse laço termina, $j = A \cdot comprimento + 1$ (ou, o que é equivalente, $j = n + 1$, visto que $n = A \cdot comprimento$).

  Usamos a palavra-chave **to** quando um laço **for** incrementa seu contador do laço a cada iteração, e usamos a palavra-chave **downto** quando um laço **for** decrementa seu contador de laço. Quando o contador do laço mudar

por uma quantidade maior do que 1, essa quantidade virá após a palavra-chave opcional **by**.

- O símbolo "//" indica que o restante da linha é um comentário.
- Uma atribuição múltipla da forma $i = j = e$ atribui às variáveis i e j o valor da expressão e; ela deve ser tratada como equivalente à atribuição $j = e$ seguida pela atribuição $i = j$.
- Variáveis (como i, j e *chave*) são locais para o procedimento dado. Não usaremos variáveis globais sem indicação explícita.
- Elementos de arranjos são acessados especificando-se o nome do arranjo seguido pelo índice entre colchetes. Por exemplo, $A[i]$ indica o i-ésimo elemento do arranjo A. A notação ".." é usada para indicar uma faixa de valores dentro de um arranjo. Desse modo, $A[1 .. j]$ indica o subarranjo de A que consiste nos j elementos $A[1]$, $A[2]$, ..., $A[j]$.
- Dados compostos estão organizados tipicamente em ***objetos***, compostos por ***atributos***. Acessamos um determinado atributo usando a sintaxe encontrada em muitas linguagens de programação orientadas a objetos: o nome do objeto, seguido por um ponto, seguido pelo nome do atributo. Por exemplo, tratamos um arranjo como um objeto com o atributo *comprimento* indicando quantos elementos ele contém. Para especificar o número de elementos em um arranjo A, escrevemos $A \cdot comprimento$.

Uma variável que representa um arranjo ou objeto é tratada como um ponteiro para os dados que representam o arranjo ou objeto. Para todos os atributos f de um objeto x, definir $y = x$ causa $y \cdot f = x \cdot f$. Além disso, se definirmos agora $x \cdot f = 3$, daí em diante não apenas $x \cdot f = 3$, mas também $y \cdot f = 3$. Em outras palavras, x e y apontarão para o mesmo objeto após a atribuição $y = x$.

A notação que usamos para atributos pode ser utilizada "em cascata". Por exemplo, suponha que o atributo f seja, em si, um ponteiro para algum tipo de objeto que tem um atributo g. Então, a notação $x \cdot f \cdot g$ estará implicitamente entre parênteses como $(x \cdot f),g$. Em outras palavras, se tivéssemos atribuído $y = x \cdot f$, então $x \cdot f \cdot g$ é o mesmo que $y \cdot g$.

Às vezes, um ponteiro não fará referência a nenhum objeto. Nesse caso, daremos a ele o valor especial NIL.

- Parâmetros são passados para um procedimento ***por valor*** : o procedimento chamado recebe sua própria cópia dos parâmetros e, se tal procedimento atribuir um valor a um parâmetro, a mudança *não* será vista pelo procedimento de chamada. Quando objetos são passados, o ponteiro para os dados que representam o objeto é copiado, mas os atributos do objeto, não. Por exemplo, se x é um parâmetro de um procedimento chamado, a atribuição $x = y$ dentro do procedimento chamado não será visível para o procedimento de chamada.
  Contudo, a atribuição $x \cdot f = 3$ será visível. De maneira semelhante, arranjos são passados por apontador; assim, um apontador para o arranjo é passado, em vez do arranjo inteiro, e as mudanças nos elementos individuais do arranjo são visíveis para o procedimento de chamada.
- Uma instrução **return** transfere imediatamente o controle de volta ao ponto de chamada no procedimento de chamada. A maioria das instruções **return** também toma um valor para passar de volta ao chamador. Nosso pseudocódigo é diferente de muitas linguagens de programação, visto que permite que vários valores sejam devolvidos em uma única instrução **return.**
- Os operadores booleanos "e" e "ou" são operadores com ***curto-circuito***. Isto é, quando avaliamos a expressão "x e y", avaliamos primeiro x. Se x for avaliado como FALSE, a expressão inteira não poderá ser avaliada como TRUE, e assim não avaliamos y. Se, por outro lado, x for avaliado como TRUE, teremos de avaliar y para determinar o valor da expressão inteira. De modo semelhante, na expressão "x ou y", avaliamos a expressão y somente se x for avaliado como FALSE. Os operadores de curto-circuito nos permitem escrever expressões booleanas como "$x \neq$ NIL e $x \cdot f = y$" sem nos preocuparmos com o que acontece ao
  tentarmos avaliar $x \cdot f$ quando x é NIL.
- A palavra-chave **error** indica que ocorreu um erro porque as condições para que o procedimento fosse chamado estavam erradas. O procedimento de chamada é responsável pelo tratamento do erro, portanto não especificamos a ação que deve ser executada.

Exercícios

2.1-1 Usando a Figura 2.2 como modelo, ilustre a operação de Insertion-Sort no arranjo $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2 Reescreva o procedimento Insertion-Sort para ordenar em ordem não crescente, em vez da ordem não decrescente.

2.1-3 Considere o problema de busca:

Entrada: Uma sequência de n números $A = \langle a_1, a_2, ..., a_n \rangle$ e um valor v.

Saída: Um índice i tal que $v = A[i]$ ou o valor especial NIL, se v não aparecer em A.

Escreva o pseudocódigo para ***busca linear***, que faça a varredura da sequência, procurando por v. Usando um invariante de laço, prove que seu algoritmo é correto. Certifique-se de que seu invariante de laço satisfaz as três propriedades necessárias.

2.1-4 Considere o problema de somar dois inteiros binários de n bits, armazenados em dois arranjos de n elementos A e B. A soma dos dois inteiros deve ser armazenada em forma binária em um arranjo de $(n + 1)$ elementos C. Enuncie o problema formalmente e escreva o pseudocódigo para somar os dois inteiros.

2.2 ANÁLISE DE ALGORITMOS

Analisar um algoritmo significa prever os recursos de que o algoritmo necessita. Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, porém mais frequentemente é o tempo de computação que desejamos medir. Em geral, pela análise de vários algoritmos candidatos para um problema, pode-se identificar facilmente um que seja o mais eficiente. Essa análise pode indicar mais de um candidato viável, porém, em geral, podemos descartar vários algoritmos de qualidade inferior no processo.

Antes de podermos analisar um algoritmo, devemos ter um modelo da tecnologia de implementação que será usada, inclusive um modelo para os recursos dessa tecnologia e seus custos. Na maior parte deste livro, consideraremos um modelo de computação genérico de máquina de acesso aleatório (***random-access machine***, ***RAM***) com um único processador como nossa tecnologia de implementação e entenderemos que nossos algoritmos serão implementados como programas de computador. No modelo de RAM, as instruções são executadas uma após outra, sem operações concorrentes.

No sentido estrito, deveríamos definir com precisão as instruções do modelo de RAM e seus custos. Porém, isso seria tedioso e nos daria pouca percepção do projeto e da análise de algoritmos. Não obstante, devemos ter cuidado para não abusar do modelo de RAM. Por exemplo, e se uma RAM tivesse uma instrução de ordenação? Então, poderíamos ordenar com apenas uma instrução. Tal RAM seria irreal, visto que os computadores reais não têm tais instruções. Portanto, nosso guia é o modo como os computadores reais são projetados. O modelo de RAM contém instruções comumente encontradas em computadores reais: instruções aritméticas (como soma, subtração, multiplicação, divisão, resto, piso, teto), de movimentação de dados (carregar, armazenar, copiar) e de controle (desvio condicional e incondicional, chamada e retorno de sub-rotinas). Cada uma dessas instruções demora uma quantidade de tempo constante.

Os tipos de dados no modelo de RAM são inteiros e de ponto flutuante (para armazenar números reais). Embora normalmente não nos preocupemos com a precisão neste livro, em algumas aplicações a precisão é crucial. Também consideramos um limite para o tamanho de cada palavra de dados. Por exemplo, ao trabalharmos com entradas de tamanho n, em geral consideramos que os inteiros são representados por $c \lg n$ bits para alguma constante $c \geq 1$.

Exigimos $c \geq 1$ para que cada palavra possa conter o valor de n, o que nos permite indexar os elementos individuais da entrada, e c terá de ser obrigatoriamente uma constante para que o tamanho da palavra não cresça arbitrariamente. (Se o tamanho da palavra pudesse crescer arbitrariamente, seria possível armazenar enorme quantidade de dados em uma única palavra e executar operações com tudo isso em tempo constante — claramente um cenário irreal.)

Computadores reais contêm instruções que não citamos, e tais instruções representam uma área cinzenta no modelo de RAM. Por exemplo, a exponenciação é uma instrução de tempo constante? No caso geral, não; são necessárias várias instruções para calcular x^y quando x e y são números reais. Porém, em situações restritas, a exponenciação é uma operação de tempo constante. Muitos computadores têm uma instrução "deslocar para a esquerda" (*shift left*) que desloca em tempo constante os bits de um inteiro k posições para a esquerda. Na maioria dos computadores, deslocar os bits de um inteiro uma posição para a esquerda equivale a multiplicar por 2; assim, deslocar os bits k posições para a esquerda equivale a multiplicar por 2^k. Portanto, tais computadores podem calcular 2^k em uma única instrução de tempo constante deslocando o inteiro 1 k posições para a esquerda, desde que k não seja maior que o número de bits em uma palavra de computador. Procuraremos evitar essas áreas cinzentas no modelo de RAM, mas trataremos o cálculo de 2^k como uma operação de tempo constante quando k for um inteiro positivo suficientemente pequeno.

No modelo de RAM, não tentamos modelar a hierarquia da memória que é comum em computadores contemporâneos. Isto é, não modelamos caches ou memória virtual. Vários modelos computacionais tentam levar em conta os efeitos da hierarquia de memória, que às vezes são significativos em programas reais em máquinas reais. Alguns problemas neste livro examinam os efeitos da hierarquia de memória mas, em sua maioria, as análises neste livro não os considerarão.

Os modelos que incluem a hierarquia de memória são bem mais complexos que o modelo de RAM, portanto pode ser difícil utilizá-los. Além disso, as análises do modelo de RAM em geral permitem previsões excelentes do desempenho em máquinas reais.

Até mesmo a análise de um algoritmo simples no modelo de RAM pode ser um desafio. As ferramentas matemáticas exigidas podem incluir análise combinatória, teoria das probabilidades, destreza em álgebra e a capacidade de identificar os termos mais significativos em uma fórmula. Tendo em vista que o comportamento de um algoritmo pode ser diferente para cada entrada possível, precisamos de um meio para resumir esse comportamento em fórmulas simples, de fácil compreensão. Embora normalmente selecionemos apenas um único modelo de máquina para analisar determinado algoritmo, ainda estaremos diante de muitas opções na hora de decidir como expressar nossa análise. Gostaríamos de dispor de um meio de expressão que seja simples de escrever e manipular, que mostre as características importantes de requisitos de recursos de um algoritmo e que suprima os detalhes tediosos.

Análise da ordenação por inserção

O tempo despendido pelo procedimento Insertion-Sort depende da entrada: ordenar mil números demora mais que ordenar três números. Além disso, Insertion-Sort pode demorar quantidades de tempo diferentes para ordenar duas sequências de entrada do mesmo tamanho, dependendo do quanto elas já estejam ordenadas. Em geral, o tempo gasto por um algoritmo cresce com o tamanho da entrada; assim, é tradicional descrever o tempo de execução de um programa em função do tamanho de sua entrada. Para isso, precisamos definir os termos "tempo de execução" e "tamanho da entrada" com mais cuidado.

A melhor noção para ***tamanho da entrada*** depende do problema que está sendo estudado. No caso de muitos problemas, como a ordenação ou o cálculo de transformações discretas de Fourier, a medida mais natural é o *número de itens na entrada* — por exemplo, o tamanho n do arranjo para ordenação. Para muitos outros problemas, como a multiplicação de dois inteiros, a melhor medida do tamanho da entrada é o *número total de bits* necessários para representar a entrada em notação binária comum. Às vezes, é mais apropriado descrever o tamanho da entrada com dois números em vez de um. Por exemplo, se a entrada para um algoritmo é um grafo, o tamanho da entrada pode ser descrito pelos números de vértices e arestas no grafo. Indicaremos qual medida de tamanho da entrada está sendo usada com cada problema que estudarmos.

O ***tempo de execução*** de um algoritmo em determinada entrada é o número de operações primitivas ou "passos" executados. É conveniente definir a noção de passo de modo que ela seja tão independente de máquina quanto possível. Por enquanto, vamos adotar a visão a seguir. Uma quantidade de tempo constante é exigida para executar cada linha do nosso pseudo código. Uma linha pode demorar uma quantidade de tempo diferente de outra linha, mas consideraremos que cada execução da *i*-ésima linha leva um tempo c_i, onde c_i é uma constante. Esse ponto de vista está de acordo com o modelo de RAM e também reflete o modo como o pseudo-código seria implementado na maioria dos computadores reais.[5]

Na discussão a seguir, nossa expressão para o tempo de execução de Insertion-Sort evoluirá de uma fórmula confusa que utiliza todos os custos de instrução c_i até uma notação muito mais simples, que também é mais concisa e mais fácil de manipular. Essa notação mais simples também facilitará a tarefa de determinar se um algoritmo é mais eficiente que outro.

Começaremos apresentando o procedimento Insertion-Sort com o "custo" de tempo de cada instrução e o número de vezes que cada instrução é executada. Para cada $j = 2, 3, ..., n$, onde $n = A \cdot comprimento$, seja t_j o número de vezes que o teste do laço **while** na linha 5 é executado para aquele valor de j. Quando um laço **for** ou **while** termina da maneira usual (isto é, devido ao teste no cabeçalho do laço), o teste é executado uma vez mais do que o corpo do laço. Consideramos que comentários não são instruções executáveis e, portanto, não demandam nenhum tempo.

Insertion-Sort(A)	*custo*	*vezes*
1 **for** $j = 2$ **to** $A \cdot comprimento$	c_1	n
2 $chave = A[j]$	c_2	$n-1$
3 //Inserir $A[j]$ na sequência ordenada $A[1..j-1]$.	0	$n-1$
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ e $A[i] > chave$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = chave$	c_8	$n-1$

O tempo de execução do algoritmo é a soma dos tempos de execução para cada instrução executada; uma instrução que demanda c_i passos para ser executada e é executada n vezes contribuirá com c_in para o tempo de execução total.[6] Para calcular $T(n)$, o tempo de execução de Insertion-Sort de uma entrada de n valores, somamos os produtos das *colunas custo* e *vezes*, obtendo

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} tj + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

Mesmo para entradas de dado tamanho, o tempo de execução de um algoritmo pode depender de *qual* entrada desse tamanho é dada. Por exemplo, em Insertion-Sort, o melhor caso ocorre se o arranjo já está ordenado. Então, para cada $j = 2, 3, ..., n$, descobrimos que $A[i] \leq chave$ na linha 5 quando i tem seu valor inicial $j - 1$. Portanto, $t_j = 1$ para $j = 2, 3, ..., n$, e o tempo de execução do melhor caso é

$$T(n) = c_{1n} + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Podemos expressar esse tempo de execução como $an + b$ para *constantes* a e b que dependem dos custos de instrução c_i; assim, ele é uma ***função linear*** de n.

Se o arranjo estiver ordenado em ordem inversa — ou seja, em ordem decrescente —, resulta o pior caso. Devemos comparar cada elemento $A[j]$ com cada elemento do subarranjo ordenado inteiro, $A[1 \mathinner{..} j-1]$, e então $t_j = j$ para 2, 3, ..., n. Observando que

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

e

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(o Apêndice A apresenta modos de resolver esses somatórios), descobrimos que, no pior caso, o tempo de execução de Insertion-Sort é

$$\begin{aligned} T(n) = {} & c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ & + c_6\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ & -(c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Podemos expressar esse tempo de execução do pior caso como $an^2 + bn + c$ para constantes a, b e c que, mais uma vez, dependem dos custos de instrução c_i; portanto, ele é uma ***função quadrática*** de n.

Em geral, como na ordenação por inserção, o tempo de execução de um algoritmo é fixo para determinada entrada, embora em capítulos posteriores veremos alguns algoritmos "aleatorizados" interessantes, cujo comportamento pode variar até mesmo para uma entrada fixa.

Análise do pior caso e do caso médio

Em nossa análise da ordenação por inserção, examinamos tanto o melhor caso, no qual o arranjo de entrada já estava ordenado, quanto o pior caso, no qual o arranjo de entrada estava ordenado em ordem inversa. Porém, no restante deste livro, em geral nos concentraremos em determinar apenas o ***tempo de execução do pior caso***; ou seja, o tempo de execução mais longo para *qualquer* entrada de tamanho n. Apresentamos três razões para essa orientação.

- O tempo de execução do pior caso de um algoritmo estabelece um limite superior para o tempo de execução para qualquer entrada. Conhecê-lo nos dá uma garantia de que o algoritmo nunca demorará mais do que esse tempo. Não precisamos fazer nenhuma suposição sobre o tempo de execução esperando que ele nunca seja muito pior.
- Para alguns algoritmos, o pior caso ocorre com bastante frequência. Por exemplo, na pesquisa de um banco de dados em busca de determinada informação, o pior caso do algoritmo de busca frequentemente ocorre quando a informação não está presente no banco de dados. Em algumas aplicações, a busca de informações ausentes pode ser frequente.
- Muitas vezes, o "caso médio" é quase tão ruim quanto o pior caso. Suponha que escolhemos n números aleatoriamente e aplicamos ordenação por inserção. Quanto tempo transcorrerá até que o algoritmo determine o lugar no subarranjo $A[1 \mathinner{..} j-1]$ em que deve ser inserido o elemento $A[j]$? Em média, metade dos elementos em $A[1 \mathinner{..} j-1]$ é menor que $A[j]$ e metade dos elementos é maior. Portanto, em média, verificamos metade do

subarranjo $A[1 .. j - 1]$ e, portanto, $t_j = j/2$. Resulta que o tempo de execução obtido para o caso médio é uma função quadrática do tamanho da entrada, exatamente o que ocorre com o tempo de execução do pior caso.

Em alguns casos particulares, estaremos interessados no tempo de execução do ***caso médio*** de um algoritmo; veremos, neste livro, a técnica da ***análise probabilística*** aplicada a vários algoritmos. O escopo da análise do caso médio é limitado porque pode não ser evidente o que constitui uma entrada "média" para determinado problema. Muitas vezes consideraremos que todas as entradas de um dado tamanho são igualmente prováveis. Na prática, é possível que essa suposição seja violada, mas, às vezes, podemos utilizar um ***algoritmo aleatorizado***, que efetua escolhas ao acaso, para permitir uma análise probabilística e produzir um tempo ***esperado*** de execução**.** Estudaremos algoritmos randomizados com mais detalhes no Capítulo 5 e em vários outros capítulos subsequentes.

Ordem de crescimento

Usamos algumas abstrações simplificadoras para facilitar nossa análise do procedimento Insertion-Sort. Primeiro, ignoramos o custo real de cada instrução, usando as constantes c_i para representar esses custos. Então, observamos que até mesmo essas constantes nos dão mais detalhes do que realmente necessitamos: expressamos o tempo de execução do pior caso como $an^2 + bn + c$ para algumas constantes a, b e c que dependem dos custos de instrução c_i. Desse modo, ignoramos não apenas os custos reais de instrução, mas também os custos abstratos c_i.

Agora, faremos mais uma abstração simplificadora. É a ***taxa de crescimento***, ou ***ordem de crescimento***, do tempo de execução que realmente nos interessa. Portanto, consideramos apenas o termo inicial de uma fórmula (por exemplo, an^2), já que os termos de ordem mais baixa são relativamente insignificantes para grandes valores de n. Também ignoramos o coeficiente constante do termo inicial, visto que fatores constantes são menos significativos que a taxa de crescimento na determinação da eficiência computacional para grandes entradas. No caso da ordenação por inserção, quando ignoramos os termos de ordem mais baixa e o coeficiente constante do termo inicial, resta apenas o fator de n^2 do termo inicial. Afirmamos que a ordenação por inserção tem um tempo de execução do pior caso igual a $\Theta(n^2)$ (lido como "teta de n ao quadrado"). Neste capítulo usaremos informalmente a notação Θ e a definiremos com precisão no Capítulo 3.

Em geral, consideramos que um algoritmo é mais eficiente que outro se seu tempo de execução do pior caso apresentar uma ordem de crescimento mais baixa. Devido a fatores constantes e termos de ordem mais baixa, um algoritmo cujo tempo de execução tenha uma ordem de crescimento mais alta pode demorar menos tempo para pequenas entradas do que um algoritmo cuja ordem de crescimento seja mais baixa. Porém, para entradas suficientemente grandes, um algoritmo $\Theta(n^2)$, por exemplo, será executado mais rapidamente no pior caso que um algoritmo $\Theta(n^3)$.

Exercícios

2.2-1 Expresse a função $n^3/1000 - 100n^2 - 100n + 3$ em termos da notação Θ.

2.2-2 Considere a ordenação de n números armazenados no arranjo A, localizando primeiro o menor elemento de A e permutando esse elemento com o elemento contido em $A[1]$. Em seguida, determine o segundo menor elemento de A e permute-o com $A[2]$. Continue dessa maneira para os primeiros $n - 1$ elementos de A. Escreva o pseudocódigo para esse algoritmo, conhecido como **ordenação por seleção**. Qual invariante de laço esse algoritmo mantém? Por que ele só precisa ser executado para os primeiros $n - 1$ elementos, e não para todos os n elementos? Forneça os tempos de execução do melhor caso e do pior caso da ordenação por seleção em notação Θ.

2.2-3 Considere mais uma vez a busca linear (veja Exercício 2.1-3). Quantos elementos da sequência de entrada precisam ser verificados em média, considerando que o elemento que está sendo procurado tenha a mesma probabilidade de ser qualquer elemento no arranjo? E no pior caso? Quais são os tempos de execução do

caso médio e do pior caso da busca linear em notação Θ? Justifique suas respostas.

2.2-4 Como podemos modificar praticamente qualquer algoritmo para ter um bom tempo de execução no melhor caso?

2.3 PROJETO DE ALGORITMOS

Há uma grande variedade de técnicas de projeto de algoritmos à nossa disposição. Para a ordenação por inserção utilizamos uma abordagem ***incremental***: tendo ordenado o subarranjo $A[1 \mathinner{..} j-1]$, inserimos o elemento isolado $A[j]$ em seu lugar apropriado, o que produz o subarranjo ordenado $A[1 \mathinner{..} j]$.

Nesta seção, examinaremos uma abordagem de projeto alternativa, conhecida como "divisão e conquista", que estudaremos com mais detalhes no Capítulo 4. Usaremos tal abordagem para projetar um algoritmo de ordenação cujo tempo de execução do pior caso é muito menor que o da ordenação por inserção. Uma vantagem dos algoritmos de divisão e conquista é que seus tempos de execução são frequentemente fáceis de determinar com a utilização de técnicas que serão apresentadas no Capítulo 4.

2.3.1 A ABORDAGEM DE DIVISÃO E CONQUISTA

Muitos algoritmos úteis são ***recursivos*** em sua estrutura: para resolver um dado problema, eles chamam a si mesmos recursivamente uma ou mais vezes para lidar com subproblemas intimamente relacionados. Em geral, esses algoritmos seguem uma abordagem de ***divisão e conquista***: eles desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas de menor tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

O paradigma de divisão e conquista envolve três passos em cada nível da recursão:

Divisão do problema em determinado número de subproblemas que são instâncias menores do problema original.

Conquista os subproblemas, resolvendo-os recursivamente. Porém, se os tamanhos dos sub-problemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta.

Combinação as soluções dadas aos subproblemas na solução para o problema original.

O algoritmo de ***ordenação por intercalação*** a seguir obedece rigorosamente ao paradigma de divisão e conquista. Intuitivamente, ele funciona do modo ilustrado a seguir.

Divisão: Divide a sequência de n elementos que deve ser ordenada em duas subsequências de $n/2$ elementos cada uma.

Conquista: Ordena as duas subsequências recursivamente, utilizando a ordenação por intercalação.

Combinação: Intercala as duas subsequências ordenadas para produzir a resposta ordenada.

A recursão "extingue-se" quando a sequência a ser ordenada tiver comprimento 1, visto que nesse caso não há nenhum trabalho a ser feito, já que toda sequência de comprimento 1 já está ordenada.

A operação-chave do algoritmo de ordenação por intercalação é a intercalação de duas sequências ordenadas, no passo de "combinação". Para executar a intercalação, chamamos um procedimento auxiliar MERGE(A, p, q, r), onde A é um arranjo e p, q e r são índices de enumeração dos elementos do arranjo, tais que $p \leq q < r$. O procedimento considera que os subarranjos $A[p \mathinner{..} q]$ e $A[q+1 \mathinner{..} r]$ estão em sequência ordenada. Ele os ***intercala*** (ou ***mescla***) para formar um único subarranjo ordenado que substitui o subarranjo atual $A[p \mathinner{..} r]$.

Nosso procedimento MERGE leva o tempo $\Theta(n)$, onde $n = r - p + 1$ é o número total de elementos que estão sendo intercalados, e funciona como descrito a seguir. Retornando ao nosso exemplo do jogo de cartas, suponha que temos duas pilhas de cartas com a face para cima sobre uma mesa. Cada pilha está ordenada, com as cartas de menor valor

em cima. Desejamos juntar as duas pilhas (fazendo a intercalação) em uma única pilha de saída ordenada, que ficará com a face para baixo na mesa. Nosso passo básico consiste em escolher a menor das duas cartas superiores nas duas pilhas viradas para cima, removê-la de sua pilha (o que exporá uma nova carta superior) e colocar essa carta com a face voltada para baixo sobre a pilha de saída. Repetimos esse passo até uma pilha de entrada se esvaziar e, então, simplesmente pegamos a pilha de entrada restante e a colocamos virada para baixo sobre a pilha de saída. Em termos computacionais, cada passo básico demanda um tempo constante, já que estamos comparando apenas as duas cartas superiores. Considerando que executamos no máximo n passos básicos, a intercalação demorará um tempo $\Theta(n)$.

O pseudocódigo a seguir implementa essa ideia, mas tem uma variação que evita a necessidade de verificar se qualquer das duas pilhas está vazia em cada passo básico. Colocamos na parte inferior de cada pilha uma carta ***sentinela***, que contém um valor especial que empregamos para simplificar nosso código. Aqui, usamos ∞ como valor de sentinela de modo que, sempre que uma carta com ∞ for exposta, ela não poderá ser a menor carta, a menos que as cartas sentinelas de ambas as pilhas estejam expostas. Porém, assim que isso ocorre, todas as cartas que não são sentinelas já terão sido colocadas sobre a pilha de saída. Como sabemos com antecedência que exatamente $r - p + 1$ cartas serão colocadas sobre a pilha de saída, podemos parar após a execução desse mesmo número de etapas básicas.

```
MERGE(A, p, q, r)
1   n1 = q − p + 1
2   n2 = r − q
3   sejam L[1..n1 + 1] e R[1.. n2 + 1] novos arranjos
4   for i = 1 to n1
5      L[i] = A[p + i − 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1] = ∞
10   i = 1
11   j = 1
12   for k = p to r
13        if L[i] ≤ R[j]
14             then A[k] = L[i]
15                  i = i + 1
16             else A[k] = R[j]
17                  j = j + 1
```

Em detalhe, o procedimento MERGE funciona da maneira ilustrada a seguir. A linha 1 calcula o comprimento n_1 do subarranjo $A[p .. q]$ e a linha 2 calcula o comprimento n_2 do subarranjo $A[q + 1.. r]$. Criamos os arranjos L e R (de "*left*" e "*right*", em inglês, ou "esquerda" e "direita") de comprimentos $n_1 + 1$ e $n_2 + 1$, respectivamente, na linha 3; a posição extra em cada arranjo conterá a sentinela. O laço **for** das linhas 4 e 5 copia o subarranjo $A[p .. q]$ em $L[1 .. n_1]$, e o laço **for** das linhas 6 e 7 copia o subarranjo $A[q + 1 .. r]$ em $R[1 .. n_2]$. As linhas 8 e 9 colocam as sentinelas nas extremidades dos arranjos L e R. As linhas 10 a 17, ilustradas na Figura 2.3, executam os $r - p + 1$ passos básicos, mantendo o invariante de laço a seguir:

Figura 2.3 Operação das linhas 10 a 17 na chamada Merge(A, 9, 12, 16) quando o subarranjo $A[9 \,..\, 16]$ contém a sequência $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. Depois de copiar e inserir sentinelas, o arranjo L contém $\langle 2, 4, 5, 7, \infty \rangle$ e o arranjo R contém $(1, 2, 3, 6, \infty)$. Posições sombreadas em tom mais claro em A contêm seus valores finais, e posições sombreadas em tom mais claro em L e R contêm valores que ainda têm de ser copiados de volta em A. Juntas, as posições sombreadas em tom mais claro sempre incluem os valores contidos originalmente em $A\,[9 \,..\, 16]$, além das duas sentinelas. Posições sombreadas em tom mais escuro em A contêm valores que serão copiados, e posições em tom mais escuro em L e R contêm valores que já foram copiados de volta em A. **(a)-(h)** Os arranjos A, L e R e seus respectivos índices k, i, e j antes de cada iteração do laço das linhas 12 a 17.

No início de cada iteração do laço **for** das linhas 12 a 17, o subarranjo $A[p \,..\, k - 1]$ contém os $k - p$ menores elementos de $L[1 \,..\, n_1 + 1]$ e $R[1 \,..\, n_2 + 1]$, em sequência ordenada.
Além disso, $L[i]$ e $R[j]$ são os menores elementos de seus arranjos que não foram copiados de volta em A.

Devemos mostrar que esse invariante de laço é válido antes da primeira iteração do laço **for** das linhas 12 a 17, que cada iteração do laço mantém o invariante e que o invariante fornece uma propriedade útil para mostrar correção quando o laço termina.

Inicialização: Antes da primeira iteração do laço, temos $k = p$, de modo que o subarranjo $A[p \,..\, k - 1]$ está vazio. Esse subarranjo vazio contém os $k - p = 0$ menores elementos de L e R e, uma vez que $i = j = 1$, tanto $L[i]$ quanto $R[j]$ são os menores elementos de seus arranjos que não foram copiados de volta em A.

Manutenção: Para ver que cada iteração mantém o invariante de laço, vamos supor primeiro que $L[i] \leq R[j]$. Então, $L[i]$ é o menor elemento ainda não copiado de volta em A. Como $A[p \,..\, k - 1]$ contém os $k - p$ menores elementos, depois de a linha 14 copiar $L[i]$ em $A[k]$, o subarranjo $A[p \,..\, k]$ conterá os $k - p + 1$ menores elementos. O incremento de k (na atualização do laço **for**) e de i (na linha 15) restabelece o invariante de laço para a próxima iteração. Se, em vez disso, $L[i] > R[j]$, então as linhas 16 e 17 executam a ação apropriada para manter o invariante de laço.

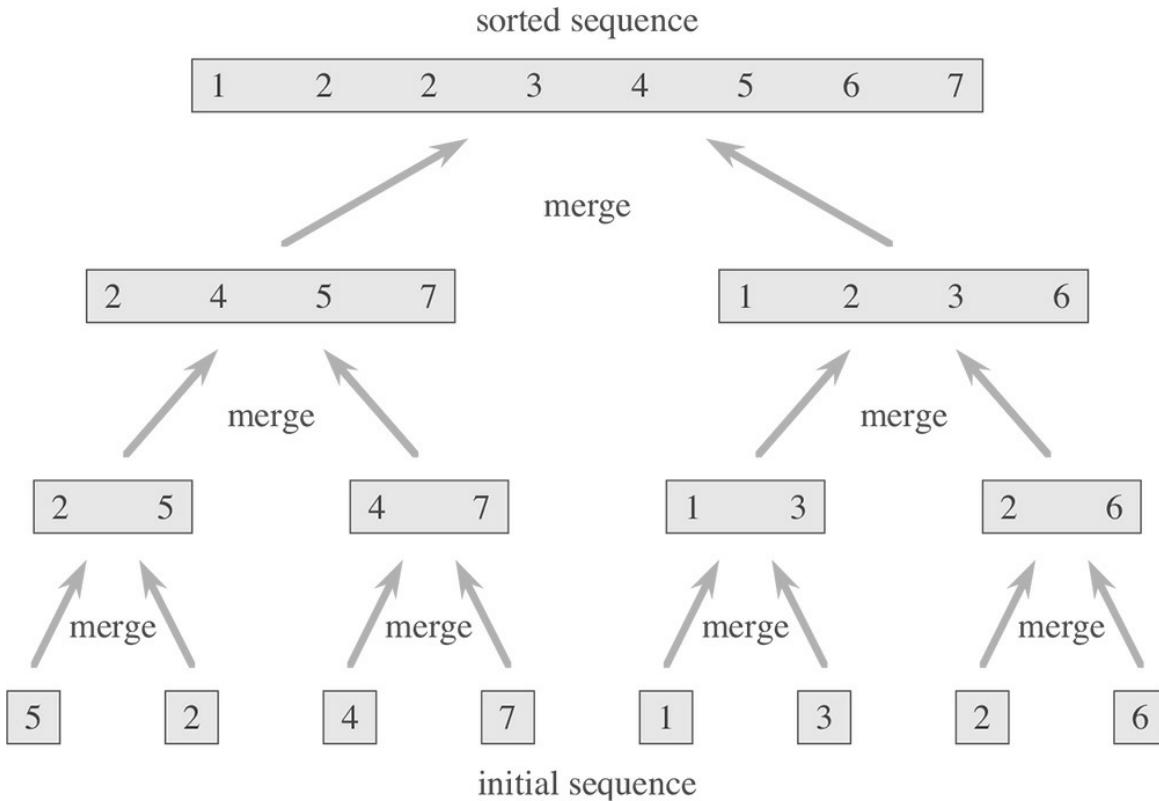
Término: No término, $k = r + 1$. Pelo invariante de laço, o subarranjo $A[p \,..\, k - 1]$, que é $A[p \,..\, r]$, contém os $k - p = r - p + 1$ menores elementos de $L[1 \,..\, n_1 + 1]$ e $R[1 \,..\, n_2 + 1]$ em sequência ordenada. Os arranjos L e R juntos contêm $n_1 + n_2 + 2 = r - p + 3$ elementos. Todos os elementos, exceto os dois maiores, foram copiados de volta em A, e esses dois maiores elementos são as sentinelas.

Para ver que o procedimento Merge é executado no tempo $\Theta(n)$, onde $n = r - p + 1$, observe que cada uma das linhas 1 a 3 e 8 a 11 demora um tempo constante, que os laços **for** das linhas 4 a 7 demoram o tempo $\Theta(n_1 + n_2) = \Theta(n)$[7] e que há n iterações do laço **for** das linhas 12 a 17, cada uma demorando um tempo constante.

Agora podemos usar o procedimento Merge como uma subrotina no algoritmo de ordenação por intercalação. O procedimento Merge-Sort(A, p, r) ordena os elementos do subarranjo $A[p \,..\, r]$. Se $p \geq r$, o subarranjo tem no máximo um elemento e, portanto, já está ordenado. Caso contrário, a etapa de divisão simplesmente calcula um índice q que subdivide $A[p \,..\, r]$ em dois subarranjos: $A[p \,..\, q]$, contendo $n/2$ elementos, e $A[q + 1 \,..\, r]$, contendo $n/2$ elementos.[8]

```
MERGE-SORT(A, p, r)
1   if p < r
2     then q = ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

Para ordenar a sequência $A = \langle A[1], A[2], ..., A[n] \rangle$ inteira, efetuamos a chamada inicial Merge-Sort$(A, 1, A \cdot comprimento)$, onde, mais uma vez, $A \cdot comprimento = n$. A Figura 2.4 ilustra a operação do procedimento de baixo para cima quando n é uma potência de 2. O algoritmo consiste em intercalar pares de sequências com 1 item para formar sequências ordenadas de comprimento 2, intercalar pares de sequências de comprimento 2 para formar sequências ordenadas de comprimento 4, e assim por diante, até que duas sequências de comprimento $n/2$ sejam intercaladas para formar a sequência ordenada final de comprimento n.

2.3.2 ANÁLISE DE ALGORITMOS DE DIVISÃO E CONQUISTA

Quando um algoritmo contém uma chamada recursiva a si próprio, seu tempo de execução pode ser descrito frequentemente por uma ***equação de recorrência*** ou ***recorrência***, que descreve o tempo de execução global para um problema de tamanho n em termos do tempo de execução para entradas menores. Então, podemos usar ferramentas matemáticas para resolver a recorrência e estabelecer limites para o desempenho do algoritmo.

Uma recorrência para o tempo de execução de um algoritmo de divisão e conquista resulta dos três passos do paradigma básico. Como antes, consideramos $T(n)$ o tempo de execução para um problema de tamanho n. Se o tamanho do problema for pequeno o bastante, digamos $n \leq c$ para alguma constante c, a solução direta demorará um

tempo constante, que representamos por $\Theta(1)$. Vamos supor que a subdivisão que adotamos para o problema produza subproblemas, cada um deles com $1/b$ do tamanho do problema original. (No caso da ordenação por intercalação, a e b são 2, mas veremos muitos algoritmos de divisão e conquista nos quais $a \neq b$.) O algoritmo leva o tempo $T(n/b)$ para resolver um subproblema de tamanho n/b e, portanto, leva o tempo $aT(n/b)$ para resolver um número a desses problemas. Se o algoritmo levar o tempo $D(n)$ para dividir o problema em subproblemas e o tempo $C(n)$ para combinar as soluções dadas aos subproblemas na solução para o problema original, obteremos a recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c. \\ aT(n/b) + D(n) + C(n) & \text{caso contrário.} \end{cases}$$

No Capítulo 4, veremos como resolver recorrências comuns que tenham essa forma.

Figura 2.4 A operação de ordenação por intercalação sobre o arranjo $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. Os comprimentos das sequências ordenadas que estão sendo intercaladas aumentam com a progressão do algoritmo da parte inferior até a parte superior.

Análise da ordenação por intercalação

Embora o pseudocódigo para Merge-Sort funcione corretamente quando o número de elementos não é par, nossa análise baseada na recorrência será simplificada se considerarmos que o tamanho do problema original é uma potência de 2. Então, cada passo de dividir produzirá duas subsequências de tamanho exatamente $n/2$. No Capítulo 4, veremos que essa premissa não afeta a ordem de crescimento da solução para a recorrência.

Apresentamos a seguir o raciocínio usado para configurar a recorrência para $T(n)$, o tempo de execução do pior caso da ordenação por intercalação para n números. A ordenação por intercalação para um único elemento demora um tempo constante. Quando temos $n > 1$ elementos, desmembramos o tempo de execução do modo explicado a seguir.

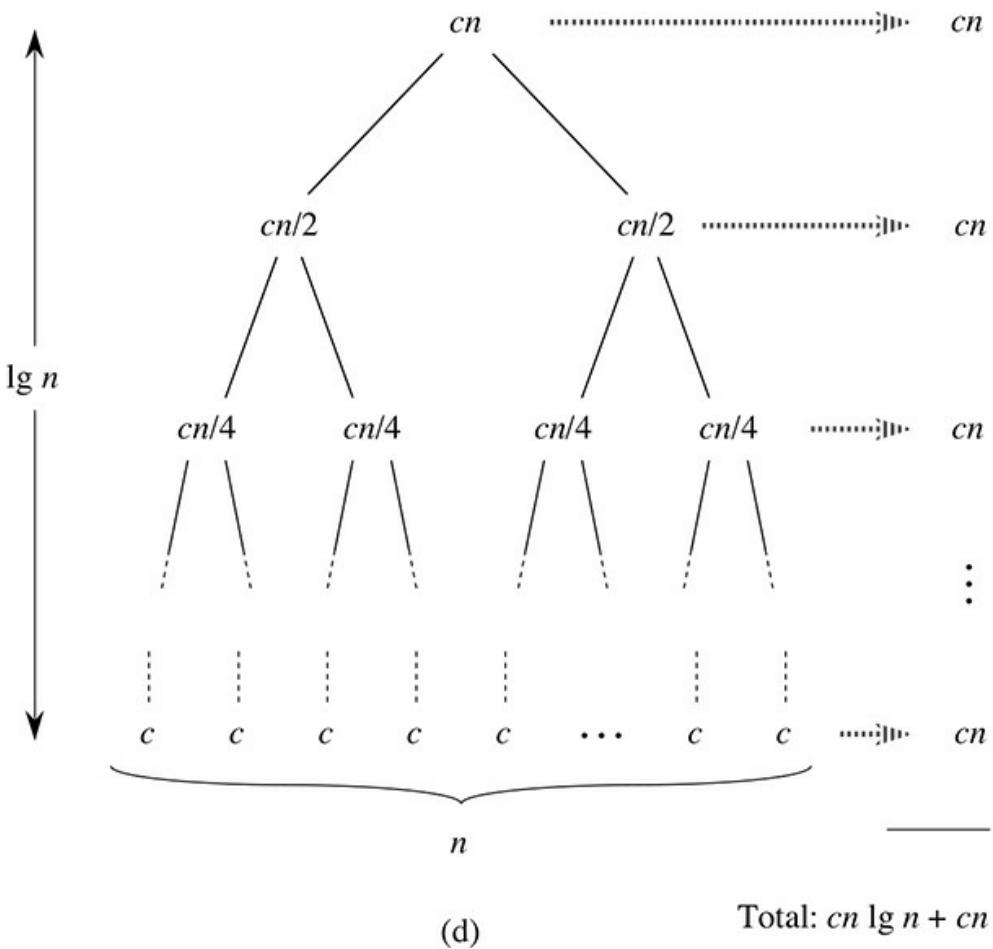
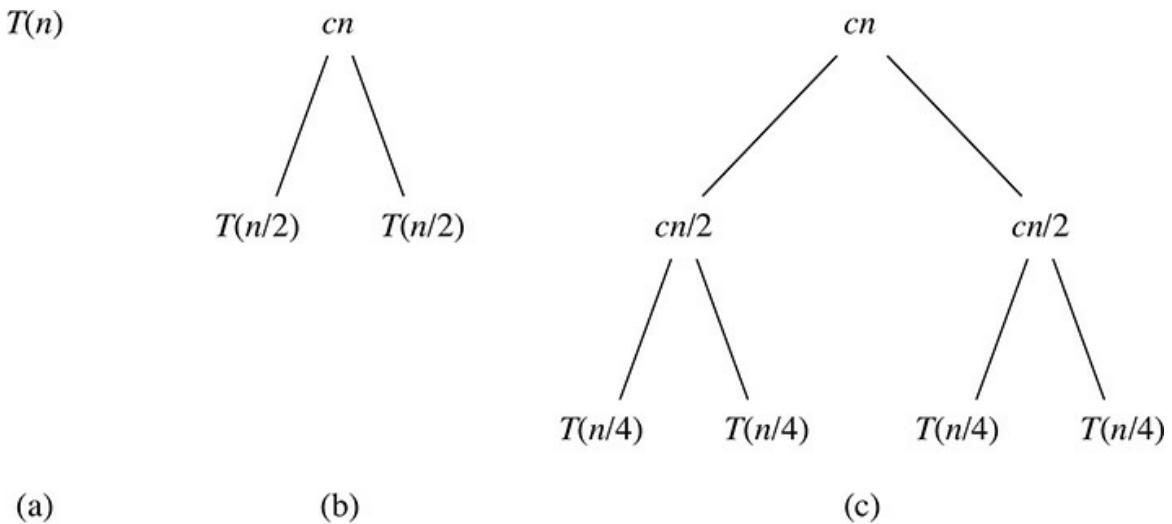
Divisão: A etapa de divisão simplesmente calcula o ponto médio do subarranjo, o que demora um tempo constante. Portanto, $D(n) = \Theta(1)$.

Conquista: Resolvemos recursivamente dois subproblemas, cada um de tamanho $n/2$, o que contribui com $2T(n/2)$ para o tempo de execução.

Combinação: Já observamos que o procedimento Merge em um subarranjo de n elementos leva o tempo $\Theta(n)$; assim, $C(n) = \Theta(n)$.

Quando somamos as funções $D(n)$ e $C(n)$ para a análise da ordenação por intercalação, estamos somando uma função que é $\Theta(n)$ a uma função que é $\Theta(1)$. Essa soma é uma função linear de n, ou seja, $\Theta(n)$. A adição dessa função ao termo $2T(n/2)$ da etapa de "conquistar" fornece a recorrência para o tempo de execução do pior caso $T(n)$ da ordenação por intercalação:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1, \\ 2T(n/2) + \Theta(n) & \text{se } n > 1. \end{cases} \quad (2.1)$$

No Capítulo 4, veremos o "teorema mestre", que podemos utilizar para mostrar que $T(n)$ é $\Theta(n \lg n)$, onde $\lg n$ significa $\log_2 n$. Como a função logarítmica cresce mais lentamente do que qualquer função linear, para entradas suficientemente grandes, o desempenho da ordenação por intercalação, com seu tempo de execução $\Theta(n \lg n)$, supera o da ordenação por inserção, cujo tempo de execução é $\Theta(n^2)$, no pior caso.

Não precisamos do teorema mestre para entender intuitivamente por que a solução para a recorrência (2.1) é $T(n) = \Theta(n \lg n)$. Vamos reescrever a recorrência (2.1) como

$$T(n) = \begin{cases} c & \text{se } n = 1, \\ 2T(n/2) + cn & \text{se } n > 1, \end{cases} \quad (2.2)$$

onde a constante c representa o tempo exigido para resolver problemas de tamanho 1, bem como o tempo por elemento do arranjo para as etapas de dividir e combinar.[9]

A Figura 2.5 mostra como podemos resolver a recorrência (2.2). Por conveniência, consideramos que n é uma potência exata de 2. A parte (a) da figura mostra $T(n)$ que, na parte (b), é expandida em uma árvore equivalente que representa a recorrência. O termo cn é a raiz (o custo incorrido no nível superior da recursão), e as duas subárvores da raiz são as duas recorrências menores $T(n/2)$. A parte (c) mostra esse processo levado uma etapa adiante pela expansão de $T(n/2)$. O custo incorrido em cada um dos dois subnós no segundo nível de recursão é $cn/2$. Continuamos a expandir cada nó na árvore, desmembrando-o em suas partes constituintes, como determinado pela recorrência, até que os tamanhos de problemas se reduzam a 1, cada qual com o custo c. A parte (d) mostra a ***árvore de recursão*** resultante.

Figura 2.5 Como construir uma árvore de recursão para a recorrência $T(n) = 2T(n/2) + cn$. A parte **(a)** mostra $T(n)$, que expande-se progressivamente em **(b)**–**(d)** para formar a árvore de recursão. A árvore completamente expandida da parte (d) tem $\lg n + 1$ níveis (isto é, tem altura $\lg n$, como indicado) e cada nível contribui com o custo total cn. Então, o custo total é $cn \lg n + cn$, que é $\Theta(n \lg n)$.

Em seguida, somamos os custos em cada nível da árvore. O nível superior tem custo total cn, o próximo nível abaixo tem custo total $c(n/2) + c(n/2) = cn$, o nível após esse tem custo total $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, e

assim por diante. Em geral, o nível i abaixo do topo tem 2^i nós, cada qual contribuindo com um custo $c(n/2^i)$, de modo que o i-ésimo nível abaixo do topo tem custo total $2^i\, c(n/2^i) = cn$. O nível inferior tem n nós, cada um contribuindo com um custo c, para um custo total cn.

O número total de níveis da árvore de recursão da Figura 2.5 é $\lg n + 1$, onde n é o número de folhas, correspondente ao tamanho da entrada. Um argumento indutivo informal justifica essa afirmação. O caso básico ocorre quando $n = 1$ e, nesse caso, a árvore tem só um nível. Como $\lg 1 = 0$, temos que $\lg n + 1$ dá o número correto de níveis. Agora suponha, como hipótese indutiva, que o número de níveis de uma árvore de recursão com 2^i folhas seja $\lg 2^i + 1 = i + 1$ (visto que, para qualquer valor de i, temos que $\lg 2^i = i$). Como estamos supondo que o tamanho da entrada é uma potência de 2, o tamanho da próxima entrada a considerar é 2^{i+1}. Uma árvore com 2^{i+1} folhas tem um nível a mais que uma árvore de 2^i folhas, e então o número total de níveis é $(i + 1) + 1 = \lg 2^{i+1} + 1$.

Para calcular o custo total representado pela recorrência (2.2), simplesmente somamos os custos de todos os níveis. A árvore de recursão tem $\lg n + 1$ níveis, cada um com custo cn, o que nos dá o custo total $cn(\lg n + 1) = cn \lg n + cn$. Ignorando o termo de ordem baixa e a constante c, obtemos o resultado desejado, $\Theta(n \lg n)$.

Exercícios

2.3-1 Usando a Figura 2.4 como modelo, ilustre a operação de ordenação por intercalação para o arranjo $A = \langle 3, 41, 52, 26, 38, 57, 9, 49\rangle$.

2.3-2 Reescreva o procedimento Merge de modo que ele não utilize sentinelas e, em vez disso, pare tão logo todos os elementos do arranjo L ou do arranjo R tenham sido copiados de volta em A e então copie o restante do outro arranjo de volta em A.

2.3-3 Use indução para mostrar que, quando n é uma potência exata de 2, a solução da recorrência

$$T(n) = \begin{cases} 2 & \text{se } n = 2, \\ 2T(n/2) + n & \text{se } n = 2^k, \text{ para } k > 1 \end{cases}$$

é $T(n) = n \lg n$.

2.3-4 A ordenação por inserção pode ser expressa como um procedimento recursivo da maneira descrita a seguir. Para ordenar $A[1 \mathinner{..} n]$, ordenamos recursivamente $A[1 \mathinner{..} n - 1]$ e depois inserimos $A[n]$ no arranjo ordenado $A[1 \mathinner{..} n - 1]$. Escreva uma recorrência para o tempo de execução de pior caso dessa versão recursiva da ordenação por inserção.

2.3-5 Voltando ao problema da busca (ver Exercício 2.1-3) observe que, se a sequência A for ordenada, poderemos comparar o ponto médio da sequência com v e não mais considerar metade da sequência. O algoritmo de ***busca binária*** repete esse procedimento, dividindo ao meio o tamanho da porção restante da sequência a cada vez. Escreva pseudocódigo, iterativo ou recursivo, para busca binária. Demonstre que o tempo de execução do pior caso da busca binária é $\Theta(\lg n)$.

2.3-6 Observe que o laço **while** das linhas 5 a 7 do procedimento Insertion-Sort na Seção 2.1 utiliza uma pesquisa linear para varrer (no sentido inverso) o subarranjo ordenado $A[1 \mathinner{..} j - 1]$. Podemos usar, em vez disso, uma busca binária (veja Exercício 2.3-5) para melhorar o tempo de execução global do pior caso da ordenação por inserção para $\Theta(n \lg n)$?

2.3-7 * Descreva um algoritmo de tempo $\Theta(n \lg n)$ que, dado um conjunto S de n inteiros e um outro inteiro x, determine se existem ou não dois elementos em S cuja soma seja exatamente x.

2.1 ***Ordenação por inserção para arranjos pequenos na ordenação por intercalação*** Embora a ordenação por intercalação funcione no tempo de pior caso $\Theta(n \lg n)$ e a ordenação por inserção funcione no tempo de pior caso $\Theta(n^2)$, os fatores constantes na ordenação por inserção podem torná-la mais rápida para n pequeno em algumas máquinas. Assim, faz sentido **adensar** as folhas da recursão usando a ordenação por inserção dentro da ordenação por intercalação quando os subproblemas se tornam suficientemente pequenos. Considere uma modificação na ordenação por intercalação, na qual n/k sublistas de comprimento k são ordenadas usando-se a ordenação por inserção e depois intercaladas com a utilização do mecanismo-padrão de intercalação, onde k é um valor a ser determinado.

a. Mostre que a ordenação por inserção pode ordenar as n/k sublistas, cada uma de comprimento k, em $\Theta(nk)$ tempo do pior caso.

b. Mostre como intercalar as sublistas em tempo do pior caso $\Theta(n \lg(n/k))$.

c. Dado que o algoritmo modificado é executado em tempo do pior caso $\Theta(nk + n \lg(n/k))$, qual é o maior valor de k em função de n para o qual o algoritmo modificado tem o mesmo tempo de execução que a ordenação por intercalação-padrão, em termos da notação Θ?

d. Como k deve ser escolhido na prática?

2.2 ***Correção do bubblesort***

O bubblesort é um algoritmo de ordenação popular, porém ineficiente. Ele funciona permutando repetidamente elementos adjacentes que estão fora de ordem.

```
BUBBLESORT(A)
1  for i = 1 to A·comprimento
2      for j = A·comprimento downto i + 1
3          if A[j] < A[j − 1]
4              then trocar A[j] com A[j − 1]
```

a. Seja A' um valor que denota a saída de Bubblesort(A). Para provar que Bubblesort é correto, precisamos provar que ele termina e que

$$A'[1] \leq A'[2] \leq \ldots \leq A'[n] , \qquad (2.3)$$

onde $n = A \cdot comprimento$. O que mais deve ser provado para mostrar que Bubblesort realmente realiza a ordenação?

As duas partes seguintes provarão a desigualdade (2.3).

b. Enuncie com precisão um invariante de laço para o laço **for** das linhas 2 a 4 e prove que esse invariante de laço é válido. Sua prova deve usar a estrutura da prova do invariante de laço apresentada neste capítulo.

c. Usando a condição de término do invariante de laço demonstrado na parte (b), enuncie um invariante de laço para o laço **for** das linhas 1 a 4 que permita provar a desigualdade (2.3). Sua prova deve empregar a estrutura da prova do invariante de laço apresentada neste capítulo.

d. Qual é o tempo de execução do pior caso de bubblesort? Como ele se compara com o tempo de execução da ordenação por inserção?

2.3 Correção da regra de Horner

O fragmento de código a seguir implementa a regra de Horner para avaliar um polinômio

$$P(x) = \sum_{k=0}^{n} a_k x^k$$
$$= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)\cdots)),$$

dados os coeficientes a_0, a_1, ..., a_n e um valor para x:

```
1  y = 0
2  for i = n downto 0
3        y = a_i + x · y
```

a. Qual é o tempo de execução desse fragmento de código em termos da notação Θ para a regra de Horner?

b. Escreva pseudocódigo para implementar o algoritmo ingênuo de avaliação polinomial que calcula cada termo do polinômio desde o início. Qual é o tempo de execução desse algoritmo? Como ele se compara com a regra de Horner?

c. Considere o seguinte invariante de laço:

No início de cada iteração do laço **for** nas linhas 2−3,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k .$$

Interprete como igual a zero um somatório que não tenha nenhum termo. Seguindo a estrutura do invariante de laço apresentado neste capítulo, use esse invariante de laço para mostrar que, no término, $y = \sum_{k=0}^{n} a_k x^k$.

d. Conclua demonstrando que o fragmento de código dado avalia corretamente um polinômio caracterizado pelos coeficientes a_0, a_1, ..., a_n.

2.4 Inversões

Seja $A[1..n]$ um arranjo de n números distintos. Se $i < j$ e $A[i] > A[j]$, então o par (i, j) é denominado ***inversão*** de A.

a. Apresente uma lista com as cinco inversões do arranjo ⟨2, 3, 8, 6, 1⟩.

b. Qual arranjo com elementos do conjunto {1, 2, ..., n} tem o maior número de inversões? Quantas inversões ele tem?

c. Qual é a relação entre o tempo de execução da ordenação por inserção e o número de inversões no arranjo de entrada? Justifique sua resposta.

d. Dê um algoritmo que determine o número de inversões em qualquer permutação com os n elementos em tempo do pior caso $\Theta(n \lg n)$. (Sugestão: Modifique a ordenação por intercalação.)

NOTAS DO CAPÍTULO

Em 1968, Knuth publicou o primeiro de três volumes com o título geral *The Art of Computer Programming* [209, 210, 211]. O primeiro volume introduziu o estudo moderno de algoritmos de computação com foco na análise do tempo de execução, e a série inteira continua a ser uma referência valiosa e interessante para muitos dos tópicos apresentados aqui. De acordo com Knuth, a palavra "algoritmo" é derivada do nome "al-Khowârizmî", um matemático persa do século IX.

Aho, Hopcroft e Ullman [5] divulgaram a análise assintótica dos algoritmos — utilizando notações apresentadas no Capítulo 3, incluindo a notação Θ — como meio de comparar o desempenho relativo. Eles também popularizaram a utilização de relações de recorrência para descrever os tempos de execução de algoritmos recursivos.

Knuth [211] oferece um tratamento enciclopédico de muitos algoritmos de ordenação. Sua comparação de algoritmos de ordenação (página 381) inclui análises exatas de contagem de passos, como a que realizamos aqui para a ordenação por inserção. A discussão por Knuth da ordenação por inserção engloba diversas variações do algoritmo. A mais importante delas é a ordenação de Shell, introduzida por D. L. Shell, que utiliza a ordenação por inserção em subsequências periódicas da entrada para produzir um algoritmo de ordenação mais rápido.

A ordenação por intercalação também é descrita por Knuth. Ele menciona que um cotejador (comparador) mecânico capaz de intercalar dois conjuntos de cartões perfurados em uma única passagem foi inventado em 1938. J. von Neumann, um dos pioneiros da ciência da computação, aparentemente escreveu um programa para ordenação por intercalação no computador EDVAC em 1945.

A história da prova de correção de programas é descrita por Gries [153], que credita a P. Naur o primeiro artigo nessa área. Gries atribui invariantes de laço a R. W. Floyd. O livro de Mitchell [256] descreve o progresso mais recente na prova da correção de programas.

---

[1] Quando o laço for um laço for, o momento no qual verificamos o invariante de laço logo antes da primeira iteração é imediatamente após a atribuição inicial à variável do contador do laço e logo antes do primeiro teste no cabeçalho do laço. No caso de Insertion-Sort, esse tempo é após atribuir 2 à variável j mas antes do primeiro teste de $j \leq A$.comprimento.

[2] Em uma instrução **if-else**, o recuo de **else** estará no mesmo nível do **if** que a acompanha. Embora a palavra-chave **then** seja omitida, ocasionalmente nos referiremos à porção executada quando o teste que vem após **if** for válido como uma ***cláusula then***. Para testes múltiplos, utilizaremos **elseif** para os testes que vêm após o primeiro.

[3] Cada procedimento de pseudocódigo neste livro aparece em uma única página; assim, você não terá que discernir níveis de identação em códigos que ocupam mais de uma página (referente à edição original).

[4] A maioria das linguagens estruturadas em blocos tem constructos equivalentes, embora a sintaxe exata possa ser diferente. A linguagem Python não tem laços **repeat-until**, e seus laços **for** funcionam de um modo um pouco diferente dos laços **for** utilizados neste livro.

[5] Há algumas sutilezas aqui. As etapas computacionais que especificamos em linguagem comum frequentemente são variantes de um procedimento que exige mais que apenas uma quantidade constante de tempo. Por exemplo, mais adiante neste livro poderíamos dizer "ordene os pontos pela coordenada x" que, como veremos, demora mais que uma quantidade constante de tempo. Além disso, observe que uma instrução que chama uma sub-rotina demora um tempo constante, embora a sub-rotina, uma vez invocada, possa durar mais. Ou seja, separamos o processo de ***chamar*** a sub-rotina — passar parâmetros a ela etc. — do processo de ***executar*** a sub-rotina.

[6] Essa característica não se mantém necessariamente para um recurso como memória. Uma instrução que referencia m palavras de memória e é executada n vezes não referencia necessariamente mn palavras de memória distintas.

[7] Veremos, no Capítulo 3, como interpretar formalmente equações contendo notação Θ.

[8] A expressão x denota o menor inteiro maior ou igual a x, e x denota o maior inteiro menor ou igual a x. Essas notações são definidas no Capítulo 3. O modo mais fácil para verificar que definir q como $(p + r)/2$ produz os subarranjos $A[p .. q]$ e $A[q + 1 .. r]$ de tamanhos $n/2$ e $n/2$, respectivamente, é examinar os quatro casos que surgem dependendo de cada valor de p e r ser ímpar ou par.

[9] É improvável que a mesma constante represente exatamente o tempo para resolver problemas de tamanho 1 e também o tempo por elemento do arranjo para as etapas de dividir e combinar. Podemos contornar esse problema tomando c como o maior desses tempos e

entendendo que nossa recorrência impõe um limite superior ao tempo de execução, ou tomando c como o menor desses tempos e entendendo que nossa recorrência impõe um limite inferior ao tempo de execução. Ambos os limites valem para a ordem de $n \lg n$ e, tomados juntos, corresponderão ao tempo de execução $\Theta(n \lg n)$.

3 CRESCIMENTO DE FUNÇÕES

A ordem de crescimento do tempo de execução de um algoritmo, definida no Capítulo 2, dá uma caracterização simples da eficiência do algoritmo e também nos permite comparar o desempenho relativo de algoritmos alternativos. Tão logo o tamanho da entrada n se torne suficientemente grande, a ordenação por intercalação, com seu tempo de execução do pior caso $\Theta(n \lg n)$, vence a ordenação por inserção, cujo tempo de execução do pior caso é $\Theta(n_2)$. Embora, às vezes, seja possível determinar o tempo exato de execução de um algoritmo, como fizemos no caso da ordenação por inserção no Capítulo 2, o que ganhamos em precisão em geral não vale o esforço do cálculo. Para entradas suficientemente grandes, as constantes multiplicativas e os termos de ordem mais baixa de um tempo de execução exato são dominados pelos efeitos do próprio tamanho da entrada. Quando observamos tamanhos de entrada suficientemente grandes para tornar relevante apenas a ordem de crescimento do tempo de execução, estamos estudando a eficiência ***assintótica*** dos algoritmos. Isto é, estamos preocupados com o modo como o tempo de execução de um algoritmo aumenta com o tamanho da entrada *no limite*, à medida que o tamanho da entrada aumenta sem limitação. Em geral, um algoritmo que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, exceto as muito pequenas.

Este capítulo oferece vários métodos padrões para simplificar a análise assintótica de algoritmos. A próxima seção começa definindo diversos tipos de "notação assintótica", da qual já vimos um exemplo na notação Θ. Então, apresentaremos várias convenções de notação usadas em todo este livro e, por fim, faremos uma revisão do comportamento de funções que surgem comumente na análise de algoritmos.

3.1 NOTAÇÃO ASSINTÓTICA

As notações que usamos para descrever o tempo de execução assintótico de um algoritmo são definidas em termos de funções cujos domínios são o conjunto dos números naturais $= \{0, 1, 2, ...\}$. Tais notações são convenientes para descrever a função $T(n)$ do tempo de execução do pior caso, que em geral é definida somente para tamanhos de entrada inteiros. Contudo, às vezes, consideramos que é conveniente *abusar* da notação assintótica de vários modos. Por exemplo, poderíamos estender a notação ao domínio dos números reais ou, como alternativa, restringi-la a um subconjunto dos números naturais. Porém, é importante entender o significado preciso da notação para que, quando abusarmos, ela não seja *mal utilizada*. Esta seção define as notações assintóticas básicas e também apresenta alguns abusos comuns.

Notação assintótica, funções e tempos de execução

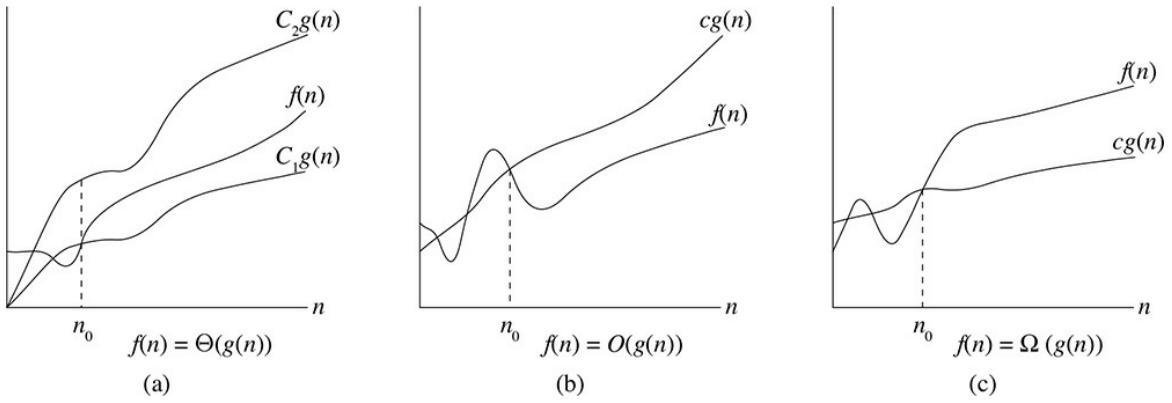
Usaremos a notação assintótica primariamente para descrever o tempo de execução de algoritmos, como fizemos quando escrevemos que o tempo de execução do pior caso para a ordenação por inserção é $\Theta(n_2)$. Todavia, na realidade a notação assintótica aplica-se a funções. Lembre-se de que caracterizamos o tempo de execução do pior caso da ordenação por inserção como $an_2 + bn + c$, para algumas constantes a, b e c. Quando afirmamos que o tempo

de execução da ordenação por inserção é $\Theta(n_2)$, abstraímos alguns detalhes dessa função. Como a notação assintótica aplica-se a funções, o que quisemos dizer é que $\Theta(n_2)$ era a função $an_2 + bn + c$ que, aqui, por acaso caracteriza o tempo de execução do pior caso da ordenação por inserção. Neste livro, as funções às quais aplicamos a notação assintótica, normalmente caracterizarão os tempos de execução de algoritmos. Porém, a notação assintótica pode se aplicar a funções que caracterizam algum outro aspecto dos algoritmos (a quantidade de espaço que eles usam, por exemplo) ou até mesmo a funções que absolutamente nada têm a ver com algoritmos.

Mesmo quando utilizamos a notação assintótica para o tempo de execução de um algoritmo, precisamos entender *a qual* tempo de execução estamos nos referindo. Às vezes, estamos interessados no tempo de execução do pior caso. Porém, frequentemente queremos caracterizar o tempo de execução, seja qual for a entrada. Em outras palavras, muitas vezes desejamos propor um enunciado abrangente que se aplique a todas as entradas, e não apenas ao pior caso. Veremos que as notações assintóticas prestam-se bem à caracterização de tempos de execução, não importando qual seja a entrada.

Notação Θ

No Capítulo 2, vimos que o tempo de execução do pior caso da ordenação por inserção é $T(n) = \Theta(n_2)$. Vamos definir o que significa essa notação. Para uma dada função $g(n)$, denotamos por $\Theta(g(n))$ o *conjunto de funções*

$$\Theta(g(n)) = \{ f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0 \}.$$ [1]

Uma função $f(n)$ pertence ao conjunto $\Theta(g(n))$ se existirem constantes positivas c_1 e c_2 tais que ela possa ser "encaixada" entre $c_1 g(n)$ e $c_2 g(n)$, para um valor de n suficientemente grande. Como $\Theta(g(n))$ é um conjunto, poderíamos escrever "$f(n) \in \Theta(g(n))$" para indicar que $f(n)$ é um membro de (ou pertence a) $\Theta(g(n))$. Em vez disso, em geral escreveremos "$f(n) = \Theta(g(n))$" para expressar a mesma noção. Esse abuso da igualdade para denotar a condição de membro de um conjunto (pertinência) pode parecer confuso, mas veremos mais adiante nesta seção que ele tem suas vantagens.

A Figura 3.1(a) apresenta um quadro intuitivo de funções $f(n)$ e $g(n)$, onde $f(n) = \Theta(g(n))$. Para todos os valores de n em n_0 ou à direita de n_0, o valor de $f(n)$ encontra-se em $c_1 g(n)$ ou acima dele e em $c_2 g(n)$ ou abaixo desse valor. Em outras palavras, para todo $n \geq n_0$, a função $f(n)$ é igual a $g(n)$ dentro de um fator constante. Dizemos que $g(n)$ é um ***limite assintoticamente restrito*** para $f(n)$.

A definição de $\Theta(g(n))$ exige que todo membro $f(n) \in \Theta(g(n))$ seja ***assintoticamente não negativo***, isto é, que $f(n)$ seja não negativa sempre que n for suficientemente grande. (Uma função ***assintoticamente positiva*** é uma função positiva para todo n suficientemente grande.) Por consequência, a própria função $g(n)$ deve ser assintoticamente não negativa, senão o conjunto $\Theta(g(n))$ é vazio. Por isso, consideraremos que toda função usada dentro da notação Θ é assintoticamente não negativa. Essa premissa também se mantém para as outras notações assintóticas definidas neste capítulo.

No Capítulo 2, introduzimos uma noção informal da notação Θ que consistia em descartar os termos de ordem mais baixa e ignorar o coeficiente inicial do termo de ordem mais alta. Vamos justificar brevemente essa intuição, usando a definição formal para mostrar que $\frac{1}{2} n_2 - 3n = \Theta(n_2)$. Para isso, devemos definir constantes positivas c , c e n tais que

$$c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$$

Figura 3.1 Exemplos gráficos das notações Θ , O e Ω . Em cada parte, o valor de n_0 mostrado é o mínimo valor possível; qualquer valor maior também funcionaria. **(a)** A notação Θ limita uma função entre fatores constantes. Escrevemos $f(n) = \Theta(g(n))$ se existirem constantes positivas n_0, c_1 e c_2 tais que, em n_0 e à direita de n_0 , o valor de $f(n)$ sempre encontrar-se entre $c_1\, g(n)$ e $c_2\, g(n)$ inclusive. **(b)** A notação O dá um limite superior para uma função dentro de um fator constante. Escrevemos $f(n) = O(g(n))$ se existirem constantes positivas n_0 e c tais que, em n_0 e à direita de n_0 , o valor de $f(n)$ sempre estiver abaixo de $cg\,(n)$. **(c)** A notação Ω dá um limite inferior para uma função dentro de um fator constante. Escrevemos $f(n) = \Omega(g(n))$ se existirem constantes positivas n_0 e c tais que, em n_0 e à direita de n_0 , o valor de $f(n)$ sempre estiver acima de $cg(n)$.

para todo $n \geq n_0$. Dividindo por n^2 temos

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 .$$

A desigualdade do lado direito pode ser considerada válida para qualquer valor de $n \geq 1$, se escolhermos qualquer constante $c_2 \geq 1/2$. Do mesmo modo, a desigualdade da esquerda pode ser considerada válida para qualquer valor de $n \geq 7$, se escolhermos qualquer constante $c_1 \leq 1/14$. Assim, escolhendo $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, podemos verificar que $\frac{1}{2}\, n2 - 3n = \Theta(n2\,)$. Certamente, existem outras opções para as constantes, mas o importante é que existe *alguma* opção. Observe que essas constantes dependem da função $\frac{1}{2}\, n2\, -3n$; uma função diferente pertencente a $\Theta(n_2)$ normalmente exigiria constantes diferentes.

Também podemos usar a definição formal para verificar que $6n_3 \neq \Theta(n_2)$. A título de contradição, suponha que existam c_2 e n_0 tais que $6n^3 \leq c_2 n^2$ para todo $n \geq n_0$. Mas, então, divisão por n^2 dá $n \leq c_2/6$, o que não pode ser válido para um valor de n arbitrariamente grande, já que c_2 é constante.

Intuitivamente, os termos de ordem mais baixa de uma função assintoticamente positiva podem ser ignorados na determinação de limites assintoticamente restritos porque eles são insignificantes para grandes valores de n. Quando n é grande, até uma minúscula fração do termo de ordem mais alta é suficiente para dominar os termos de ordem mais baixa. Desse modo, definir c_1 como um valor ligeiramente menor que o coeficiente do termo de ordem mais alta e definir c_2 como um valor ligeiramente maior permite que as desigualdades na definição da notação Θ sejam satisfeitas. Da mesma maneira, o coeficiente do termo de ordem mais alta pode ser ignorado, já que ele só muda c_1 e c_2 por um fator constante igual ao coeficiente.

Como exemplo, considere qualquer função quadrática $f(n) = an^2 + bn + c$, onde a, b e c são constantes e $a > 0$. Descartando os termos de ordem mais baixa e ignorando a constante, produzimos $f(n) = \Theta(n_2)$. Formalmente, para mostrar a mesma coisa, tomamos as constantes $c_1 = a/4$, $c_2 = 7a/4$ e $n_0 = 2 \cdot \max(|b|/a, \sqrt{|c|/a})$. . O leitor poderá

verificar que $0 \le c_1n^2 \le an^2 + bn + c \le c_2n^2$ para todo $n \ge n_0$. Em geral, para qualquer polinômio $p(n) = \sum_{i=0}^{d} a_i n^i$, onde a_i são constantes e $a_d > 0$, temos $p(n) = \Theta(n^d)$ (veja Problema 3-1).

Tendo em vista que qualquer constante é um polinômio de grau 0, podemos expressar qualquer função constante como $\Theta(n_0)$ ou $\Theta(1)$. Porém, esta última notação é um pequeno abuso porque a expressão não indica qual variável está tendendo a infinito.[2] Usaremos com frequência a notação $\Theta(1)$ para indicar uma constante ou uma função constante em relação a alguma variável.

Notação *O*

A notação Θ limita assintoticamente uma função acima e abaixo. Quando temos apenas um ***limite assintótico superior***, usamos a notação O. Para uma dada função $g(n)$, denotamos por $O(g(n))$ (lê-se "*Ó* grande de g de n" ou, às vezes, apenas "*ó* de g de n") o conjunto de funções

$$O(g(n)) = \{f(n): \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \le f(n) \le cg(n) \text{ para todo } n \ge n_0\}.$$

Usamos a notação O para dar um limite superior a uma função, dentro de um fator constante. A Figura 3.1(b) mostra a intuição por trás da notação O. Para todos os valores n em n_0 ou à direita de n_0, o valor da função $f(n)$ está abaixo de $cg(n)$.

Escrevemos $f(n) = O(g(n))$ para indicar que uma função $f(n)$ é um membro do conjunto $O(g(n))$. Observe que $f(n) = \Theta(g(n))$ implica $f(n) = O(g(n))$, já que a notação Θ é uma noção mais forte que a notação O. Em termos da teoria de conjuntos, escrevemos $\Theta(g(n)) \subseteq O(g(n))$. Assim, nossa prova de que qualquer função quadrática $an_2 + bn + c$, onde $a > 0$ está em $\Theta(n_2)$ também mostra que qualquer função quadrática desse tipo está em $O(n_2)$. O que pode ser mais surpreendente é que, quando $a > 0$, qualquer função *linear* $an + b$ está em $O(n_2)$, o que é facilmente verificado fazendo $c = a + |b|$ e $n_0 = \max(1, -b/a)$.

Se você já viu a notação O antes, poderá achar estranho que escrevamos, por exemplo, $n = O(n_2)$. Na literatura, verificamos que, às vezes, a notação O é utilizada informalmente para descrever limites assintoticamente justos, isto é, aquilo que definimos usando a notação Θ. Contudo, neste livro, quando escrevermos $f(n) = O(g(n))$, estaremos simplesmente afirmando que algum múltiplo constante de $g(n)$ é um limite assintótico superior para $f(n)$, sem qualquer menção de precisão. A distinção entre limites assintóticos superiores e limites assintoticamente justos é padrão na literatura de algoritmos.

Usando a notação O, podemos descrever frequentemente o tempo de execução de um algoritmo apenas inspecionando a estrutura global do algoritmo. Por exemplo, a estrutura de loop duplamente aninhado do algoritmo de ordenação por inserção vista no Capítulo 2 produz imediatamente um limite superior $O(n_2)$ para o tempo de execução do pior caso: o custo de cada iteração do loop interno é limitado na parte superior por $O(1)$ (constante), os índices i e j são no máximo n, e o loop interno é executado no máximo uma vez para cada um dos n_2 pares de valores para i e j. Tendo em vista que a notação O descreve um limite superior, quando a empregamos para limitar o tempo de execução do pior caso de um algoritmo temos um limite para o tempo de execução do algoritmo em cada entrada — o enunciado abrangente do qual falamos anteriormente. Desse modo, o limite $O(n_2)$ para o tempo de execução do pior caso da ordenação por inserção também se aplica a seu tempo de execução para toda entrada. Porém, o limite $\Theta(n_2)$ para o tempo de execução do pior caso da ordenação por inserção não implica um limite $\Theta(n_2)$ para o tempo de execução da ordenação por inserção em *toda* entrada. Por exemplo, vimos no Capítulo 2 que, quando a entrada já está ordenada, a ordenação por inserção funciona no tempo $\Theta(n)$.

Tecnicamente, é um abuso dizer que o tempo de execução da ordenação por inserção é $O(n_2)$, visto que, para um dado n, o tempo de execução real varia, dependendo da entrada específica de tamanho n. Quando afirmamos que "o tempo de execução é $O(n_2)$", queremos dizer que existe uma função $f(n)$ que é $O(n_2)$ tal que, para qualquer valor de n,

não importando qual entrada específica de tamanho n seja escolhida, o tempo de execução para essa entrada tem um limite superior determinado pelo valor $f(n)$. De modo equivalente, dizemos que o tempo de execução do pior caso é $O(n_2)$.

Notação Ω

Da mesma maneira que a notação O fornece um limite assintótico *superior* para uma função, a notação Ω nos dá um ***limite assintótico inferior***. Para uma determinada função $g(n)$, denotamos por $\Omega(g(n))$ (lê-se "ômega grande de g de n" ou, às vezes, "ômega de g de n") o conjunto de funções

$$\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq cg(n) \leq f(n) \text{ para todo } n \geq n_0\}.$$

A Figura 3.1(c) mostra a intuição por trás da notação Ω. Para todos os valores n em n_0 ou à direita de n_0, o valor de $f(n)$ encontra-se em $g(n)$ ou acima de $g(n)$.

Pelas definições das notações assintóticas que vimos até agora, é fácil demonstrar o importante teorema a seguir (veja Exercício 3.1-5).

Teorema 3.1

Para quaisquer duas funções $f(n)$ e $g(n)$, temos $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Como exemplo de aplicação desse teorema, nossa demonstração de que $an_2 + bn + c = \Theta(n_2)$ para quaisquer constantes a, b e c, onde $a > 0$, implica imediatamente que $an_2 + bn + c = \Omega(n_2)$ e $an_2 + bn + c = O(n_2)$. Na prática, em vez de usar o Teorema 3.1 para obter limites assintóticos superiores e inferiores a partir de limites assintoticamente precisos, como fizemos nesse exemplo, nós o utilizamos normalmente para demonstrar limites assintoticamente precisos a partir de limites assintóticos superiores e inferiores.

Quando dizemos que o *tempo de execução* (sem modificador) de um algoritmo é $\Omega(g(n))$, queremos dizer que, *não importando qual entrada específica de tamanho n seja escolhida para cada valor de n*, o tempo de execução para essa entrada é no mínimo uma constante vezes $g(n)$, para n suficientemente grande. De modo equivalente, estamos dando um limite inferior para o tempo de execução do melhor caso de um algoritmo. Por exemplo, o tempo de execução para o melhor caso da ordenação por inserção é $\Omega(n)$, o que implica que o tempo de execução da ordenação por inserção é $\Omega(n)$.

Portanto, o tempo de execução da ordenação por inserção pertence às funções $\Omega(n)$ e $O(n_2)$, já que ele se encontra em qualquer lugar entre uma função linear de n e uma função quadrática de n. Além disso, esses limites são tão justos assintoticamente quanto possível: por exemplo, o tempo de execução da ordenação por inserção não é $\Omega(n_2)$, visto que existe uma entrada para a qual a ordenação por inserção é executada no tempo $\Theta(n)$ (por exemplo, quando a entrada já está ordenada). Entretanto, não é contraditório dizer que o tempo de execução do *pior caso* da ordenação por inserção é $\Omega(n_2)$, visto que existe uma entrada que faz o algoritmo demorar o tempo $\Omega(n_2)$.

Notação assintótica em equações e desigualdades

Já vimos como a notação assintótica pode ser usada dentro de fórmulas matemáticas. Por exemplo, quando apresentamos a notação O, escrevemos "$n = O(n_2)$". Também poderíamos escrever $2n_2 + 3n + 1 = 2n_2 + \Theta(n)$. Como interpretaremos tais fórmulas?

Quando a notação assintótica está sozinha (isto é, não está dentro de uma fórmula maior) no lado direito de uma equação (ou desigualdade), como em $n = O(n_2)$, já definimos que o sinal de igualdade significa pertinência a um conjunto: $n \in O(n_2)$. Porém, em geral, quando a notação assintótica aparece em uma fórmula, interpretamos que ela representa alguma função anônima que não nos preocupamos em nomear. Por exemplo, a fórmula $2n_2 + 3n + 1 = 2n_2 +$

$\Theta(n)$ significa que $2n_2 + 3n + 1 = 2n_2 + f(n)$, onde $f(n)$ é alguma função no conjunto $\Theta(n)$. Nesse caso vale $f(n) = 3n + 1$, que de fato está em $\Theta(n)$.

Utilizar a notação assintótica dessa maneira pode ajudar a eliminar detalhes não essenciais e confusos em uma equação. Por exemplo, no Capítulo 2 expressamos o tempo de execução do pior caso da ordenação por intercalação como a recorrência

$$T(n) = 2T(n/2) + \Theta(n).$$

Se estivermos interessados apenas no comportamento assintótico de $T(n)$, não há sentido em especificar exatamente todos os termos de ordem mais baixa; entendemos que todos eles estão incluídos na função anônima denotada pelo termo $\Theta(n)$.

Entendemos também que o número de funções anônimas em uma expressão é igual ao número de vezes que a notação assintótica aparece. Por exemplo, na expressão

$$\sum_{i=1}^{n} O(i),$$

há apenas uma função anônima (uma função de i). Portanto, essa expressão *não* é o mesmo que $O(1) + O(2) + ... + O(n)$ que, na realidade, não tem uma interpretação clara.

Em alguns casos, a notação assintótica aparece no lado esquerdo de uma equação, como em $2n_2 + \Theta(n) = \Theta(n_2)$.

Interpretamos tais equações usando a seguinte regra: *independentemente de como as funções anônimas são escolhidas no lado esquerdo do sinal de igualdade, existe um modo de escolher as funções anônimas no lado direito do sinal de igualdade para tornar a equação válida*. Assim, nosso exemplo significa que, para *qualquer* função $f(n) \in \Theta(n)$, existe *alguma* função $g(n) \in \Theta(n_2)$, tal que $2n_2 + f(n) = g(n)$ para todo n. Em outras palavras, o lado direito de uma equação fornece um nível mais grosseiro de detalhes que o lado esquerdo.

Podemos encadear várias dessas relações, como em

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2). \end{aligned}$$

Podemos interpretar cada equação separadamente pelas regras citadas. A primeira equação diz que existe *alguma* função $f(n) \in \Theta(n)$ tal que $2n_2 + 3n + 1 = 2n_2 + f(n)$ para todo n. A segunda equação afirma que, para *qualquer* função $g(n) \in \Theta(n)$ (como a função $f(n)$ que acabamos de mencionar), existe *alguma* função $h(n) \in \Theta(n_2)$ tal que $2n_2 + g(n) = h(n)$ para todo n. Observe que essa interpretação implica que $2n_2 + 3n + 1 = \Theta(n_2)$, que é aquilo que o encadeamento de equações nos dá intuitivamente.

Notação *O*

O limite assintótico superior fornecido pela notação O pode ser ou não assintoticamente justo. O limite $2n_2 = O(n_2)$ é assintoticamente justo, mas o limite $2n = O(n_2)$ não é. Usamos a notação o para denotar um limite superior que não é assintoticamente justo. Definimos formalmente $o(g(n))$ (lê-se "*ó* pequeno de g de n") como o conjunto

$$o(g(n)) = \{f(n)\text{: para qualquer constante positiva } c > 0\text{, existe uma constante } n_0 > 0 \text{ tal que } 0 \leq f(n) < cg(n) \text{ para todo } n \geq n_0\}\ .$$

Por exemplo, $2n = o(n_2)$, mas $2n_2 \neq o(n_2)$.

As definições da notação O e da notação o são semelhantes. A principal diferença é que em $f(n) = O(g(n))$, o limite $0 \leq f(n) \leq cg(n)$ se mantém válido para *alguma* constante $c > 0$ mas, em $f(n) = o(g(n))$, o limite $0 \leq f(n) < cg(n)$

é válido para *todas* as constantes $c > 0$. Intuitivamente, na notação o, a função $f(n)$ torna-se insignificante em relação a $g(n)$ à medida que n se aproxima do infinito; isto é,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \tag{3.1}$$

Alguns autores usam esse limite como uma definição da notação o; a definição neste livro também restringe as funções anônimas a assintoticamente não negativas.

Notação ω

Por analogia, a notação ω está para a notação Ω como a notação o está para a notação O. Usamos a notação ω para denotar um limite inferior que não é assintoticamente preciso. Um modo de defini-lo é

$f(n) \in \omega(g(n))$ se e somente se $g(n) \in o(f(n))$.

Porém, formalmente, definimos $\omega(g(n))$ (lê-se "ômega pequeno de g de n") como o conjunto $\omega(g(n)) = \{ f(n)$: para qualquer constante positiva $c > 0$, existe uma constante

$n_0 > 0$ tal que $0 \leq cg(n) < f(n)$ para todo $n \geq n_0\}$.

Por exemplo, $n^2/2 = \omega(n)$, mas $n_2/2 \neq \omega(n_2)$. A relação $f(n) = \omega(g(n))$ implica que

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty,$$

se o limite existir. Isto é, $f(n)$ se torna arbitrariamente grande em relação a $g(n)$ à medida que n se aproxima do infinito.

Comparação de funções

Muitas das propriedades relacionais de números reais também se aplicam às comparações assintóticas. No caso das propriedades seguintes, considere que $f(n)$ e $g(n)$ são assintoticamente positivas.

Transitividade:

$f(n) = \Theta(g(n))$	e	$g(n) = \Theta(h(n))$	implicam	$f(n) = \Theta(h(n))$,
$f(n) = O(g(n))$	e	$g(n) = O(h(n))$	implicam	$f(n) = O(h(n))$,
$f(n) = \Omega(g(n))$	e	$g(n) = \Omega(h(n))$	implicam	$f(n) = \Omega(h(n))$,
$f(n) = o(g(n))$	e	$g(n) = o(h(n))$	implicam	$f(n) = o(h(n))$,
$f(n) = (g(n))$	e	$g(n) = (h(n))$	implicam	$f(n) = (h(n))$.

Reflexividade:

$f(n) = \Theta(f(n))$
$f(n) = O(f(n))$
$f(n) = \Omega(f(n))$

Simetria:

$f(n) = \Theta(\, g(n))$ se e somente se $g(n) = \Theta(\, f(n))$.

Simetria de transposição:

$f(n) = O(\, g(n))$ se e somente se $g(n) = \Omega(\, f(n))$,
$f(n) = o(\, g(n))$ se e somente se $g(n) = \omega(\, f(n))$.

Como essas propriedades se mantêm válidas para notações assintóticas, podemos traçar uma analogia entre a comparação assintótica de duas funções f e g e a comparação de dois números reais a e b:

$f(n) = O(\, g(n))$	é como	$a \leq b$,
$f(n) = \Omega(\, g(n))$	é como	$a \geq b$,
$f(n) = \Theta(\, g(n))$	é como	$a = b$,
$f(n) = o(\, g(n))$	é como	$a < b$,
$f(n) = (\, g(n))$	é como	$a > b$.

Dizemos que $f(n)$ é ***assintoticamente menor*** que $g(n)$ se $f(n) = o(g(n))$ e que $f(n)$ é ***assintoticamente maior*** que $g(n)$ se $f(n) = \omega(\, g(n))$.

Contudo, uma das propriedades de números reais não é transportada para a notação assintótica:

Tricotomia: Para quaisquer dois números reais a e b, exatamente uma das propriedades a seguir deve ser válida: $a < b$, $a = b$ ou $a > b$.

Embora quaisquer dois números reais possam ser comparados, nem todas as funções são assintoticamente comparáveis. Isto é, para duas funções $f(n)$ e $g(n)$, pode acontecer que nem $f(n) = O(g(n))$ nem $f(n) = \Omega(g(n))$ sejam válidas. Por exemplo, não podemos comparar as funções n e $n_1 + {}^{sen\, n}$ utilizando a notação assintótica, visto que o valor do expoente em $n_1 + {}^{sen\, n}$ oscila entre 0 e 2, assumindo todos os valores intermediários.

Exercícios

3.1-1 Sejam $f(n)$ e $g(n)$ funções assintoticamente não negativas. Usando a definição básica da notação Θ, prove que $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3.1-2 Mostre que, para quaisquer constantes reais a e b, onde $b > 0$,

$$(n + a)^b = \Theta(n^b)\ . \tag{3.2}$$

3.1-3 Explique por que a declaração "O tempo de execução no algoritmo A é no mínimo $O(n_2)$" não tem sentido.

3.1-4 É verdade que $2_n{+}{}^{1} = O(2_n)$? É verdade que $2^2{}_n = O(2_n)$?

3.1-5 Demonstre o Teorema 3.1.

3.1-6 Prove que o tempo de execução de um algoritmo é $\Theta(\, g(n))$ se e somente se seu tempo de execução do pior caso é $O(\, g(n))$ e seu tempo de execução do melhor caso é $\Omega(\, g(n))$.

3.1-7 Prove que $o(\, g(n)) \cap \omega(\, g(n))$ é o conjunto vazio.

3.1-8 Podemos estender nossa notação ao caso de dois parâmetros n e m que podem tender a infinito independentemente a taxas distintas. Para uma dada função $g(n, m)$, denotamos por $O(g(n, m))$ o conjunto de funções

$$\begin{aligned}O(g(n,m)) = \{f(n,m)\colon & \text{ existem constantes positivas } c, n_0 \text{ e } m_0 \\ & \text{ tais que } 0 \leq f(n,m) \leq cg(n,m) \\ & \text{ para todo } n \geq n_0 \text{ ou } m \geq m_0\}\,.\end{aligned}$$

Forneça definições correspondentes para $\Omega(g(n, m))$ e $\Theta(g(n, m))$.

3.2 Notações padrão e funções comuns

Esta seção revê algumas funções e notações matemáticas padrões e explora as relações entre elas. A seção também ilustra o uso das notações assintóticas.

Monotonicidade

Uma função $f(n)$ é ***monotonicamente crescente*** se $m \leq n$ implica $f(m) \leq f(n)$. De modo semelhante, ela é ***monotonicamente decrescente*** se $m \leq n$ implica $f(m) \geq f(n)$. Uma função $f(n)$ é ***estritamente crescente*** se $m < n$ implica $f(m) < f(n)$ e ***estritamente decrescente*** se $m < n$ implica $f(m) > f(n)$.

Pisos e tetos

Para qualquer número real x, denotamos o maior inteiro menor ou igual a x por x (lê-se "o piso de x") e o menor inteiro maior ou igual a x por x (lê-se "o teto de x"). Para todo x real,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1\,. \tag{3.3}$$

Para qualquer inteiro n,
$n/2 + n/2 = n$
e, para qualquer número real $n \geq 0$ e inteiros a, $b > 0$,

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil, \tag{3.4}$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor, \tag{3.5}$$

$$\left\lceil \frac{a}{b} \right\rceil \leq \frac{a+(b-1)}{b}, \tag{3.6}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \geq \frac{a-(b-1)}{b}. \tag{3.7}$$

A função piso $f(x) = x$ é monotonicamente crescente, como também a função teto $f(x) = x$.

Aritmética modular

Para qualquer inteiro a e qualquer inteiro positivo n, o valor a mod n é o ***resto*** do quociente a/n: a mod $n = a -$ a/n n. (3.8)

$$a \bmod n = a - \lfloor a/n \rfloor n. \tag{3.8}$$

Segue-se que

$$0 \leq a \bmod n < n. \tag{3.9}$$

Dada uma noção bem definida do resto da divisão de um inteiro por outro, é conveniente providenciar notação especial para indicar a igualdade de restos. Se (a mod n) = (b mod n), escrevemos $a \equiv b$ (mod n) e dizemos que a é ***equivalente*** a b , módulo n. Em outras palavras, $a \equiv b$ (mod n) se a e b têm o mesmo resto quando divididos por n. De modo equivalente, $a \equiv b$ (mod n) se e somente se n é um divisor de $b - a$. Escrevemos $a \not\equiv b$ (mod n) se a não é equivalente a b, módulo n.

Polinômios

Dado um inteiro não negativo d, um ***polinômio em n de grau d*** é uma função $p(n)$ da forma

$$p(n) = \sum_{i=0}^{d} a_i n^i.$$

onde as constantes $a_0, a_1, ..., a_d$ são os ***coeficientes*** do polinômio e $a_d \neq 0$. Um polinômio é assintoticamente positivo se e somente se $a_d > 0$. No caso de um polinômio assintoticamente positivo $p(n)$ de grau d, temos $p(n) = \Theta(n_d)$. Para qualquer constante real $a \geq 0$, a função n_a é monotonicamente crescente, e para qualquer constante real $a \leq 0$, a função n_a é monotonicamente decrescente. Dizemos que uma função $f(n)$ é ***polinomialmente limitada*** se $f(n) = O(n_k)$ para alguma constante k.

Exponenciais

Para todos os valores $a \neq 0$, m e n reais, temos as seguintes identidades:

$$\begin{aligned} a^0 &= 1, \\ a^1 &= a, \\ a^{-1} &= 1/a, \\ (a^m)^n &= a^{mn}, \\ (a^m)^n &= (a^n)^m, \\ a^m a^n &= a^{m+n}. \end{aligned}$$

Para todo n e $a \geq 1$, a função a_n é monotonicamente crescente em n. Quando conveniente, consideraremos $0^0 = 1$.

Podemos relacionar as taxas de crescimento de polinômios e exponenciais pelo fato a seguir. Para todas as constantes reais a e b tais que $a > 1$,

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0 \quad (3.10)$$

da qual podemos concluir que

$$n^b = o(a^n) .$$

Portanto, qualquer função exponencial com uma base estritamente maior que 1 cresce mais rapidamente que qualquer função polinomial.

Usando e para denotar 2,71828..., a base da função logaritmo natural, temos para todo x real,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} , \quad (3.11)$$

onde "!" denota a função fatorial, definida mais adiante nesta seção. Para todo x real, temos a desigualdade

$$e^x \geq 1 + x , \quad (3.12)$$

onde a igualdade vale somente quando $x = 0$. Quando $|x| \leq 1$, temos a aproximação

$$1 + x \leq e^x \leq 1 + x + x^2 . \quad (3.13)$$

Quando $x \to 0$, a aproximação de e_x por $1 + x$ é bastante boa:

$$e^x = 1 + x + \Theta(x^2) .$$

(Nessa equação, a notação assintótica é usada para descrever o comportamento limitante como $x \to 0$ em vez de $x \to \infty$.) Temos, para todo x,

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x \quad (3.14)$$

Logaritmos

Utilizaremos as seguintes notações:

$\lg n$	=	$\log_2 n$	(logaritmo binário),
$\ln n$	=	$\log_e n$	(logaritmo natural),
$\lg_k n$	=	$(\lg n)_k$	(exponenciação),
$\lg \lg n$	=	$\lg(\lg n)$	(composição).

Uma importante convenção de notação que adotaremos é que *funções logarítmicas se aplicarão apenas ao próximo termo na fórmula*; assim, $\lg n + k$ significará $(\lg n) + k$ e não $\lg(n + k)$. Se mantivermos $b > 1$ constante, então para $n > 0$ a função $\log_b n$ será estritamente crescente.

Para todo $a > 0$, $b > 0$, $c > 0$ e n real,

$$
\begin{aligned}
a &= b^{\log_b a}, \\
\log_c(ab) &= \log_c a + \log_c b, \\
\log_b a^n &= n \log_b a, \\
\log_b a &= \frac{\log_c a}{\log_c b}, \\
\log_b(1/a) &= -\log_b a, \\
\log_b a &= \frac{1}{\log_a b}, \\
a^{\log_b c} &= c^{\log_b a},
\end{aligned}
$$

onde, em cada uma dessas equações, nenhuma das bases de logaritmos é 1.

Pela equação (3.15), mudar a base de um logaritmo de uma constante para outra altera o valor do logaritmo somente por um fator constante, portanto usaremos frequentemente a notação "lg n" quando não nos importarmos com fatores constantes, como na notação O. Os cientistas da computação consideram que 2 é a base mais natural para logaritmos porque muitos algoritmos e estruturas de dados envolvem a divisão de um problema em duas partes.

Existe uma expansão de série simples para $\ln(1 + x)$ quando $|x| < 1$:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots.$$

Também temos as seguintes desigualdades para $x > -1$:

$$\frac{x}{1+x} \le \ln(1+x) \le x \; , \qquad (3.17)$$

onde a igualdade é válida somente para $x = 0$.

Dizemos que uma função $f(n)$ é ***polilogaritmicamente limitada*** se $f(n) = O(\lg^k n)$ para alguma constante k. Podemos relacionar o crescimento de polinômios e polilogaritmos substituindo

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0.$$

Desse limite, podemos concluir que

$\lg^b n = o(n_a)$

para qualquer constante $a > 0$. Desse modo, qualquer função polinomial positiva cresce mais rapidamente que qualquer função polilogarítmica.

Fatoriais

A notação $n!$ (lê-se "n fatorial") é definida para inteiros $n \ge 0$ como

$$n! = \begin{cases} 1 & \text{se } n = 0, \\ n \cdot (n-1)! & \text{se } n > 0. \end{cases}$$

Assim, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

Um limite superior fraco para a função fatorial é $n! \le n_n$, visto que cada um dos n termos no produto do fatorial é no máximo n. A ***aproximação de Stirling***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

onde e é a base do logaritmo natural, dá um limitante superior mais preciso, e também um limitante inferior. O Exercício 3.2-3 pede para provar:

$$\begin{aligned} n! &= o(n^n), \\ n! &= \omega(2^n), \\ \lg(n!) &= \Theta(n \lg n), \end{aligned} \qquad (3.19)$$

onde a aproximação de Stirling é útil na demonstração da equação (3.19). A equação a seguir também é válida para todo $n \ge 1$:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha n} \qquad (3.20)$$

onde

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}. \qquad (3.21)$$

Iteração funcional

Usamos a notação $f^{(i)}(n)$ para denotar a função $f(n)$ aplicada iterativamente i vezes a um valor inicial de n. Formalmente, seja $f(n)$ uma função no domínio dos números reais. Para inteiros não negativos i, definimos recursivamente:

$$f^{(i)}(n) = \begin{cases} n & \text{se } i = 0, \\ f(f^{(i-1)}(n)) & \text{se } i > 0. \end{cases}$$

Por exemplo, se $f(n) = 2n$, então $f^{(i)}(n) = 2^i n$.

A função logaritmo iterado

Usamos a notação $\lg^* n$ (lê-se "log estrela de n") para denotar o logaritmo iterado, que é definido da seguinte maneira: seja $\lg^{(i)} n$ definida da maneira anterior, com $f(n) = \lg n$. Como o logaritmo de um número não positivo é indefinido, $\lg^{(i)} n$ só é definido se $\lg^{(i-1)} n > 0$. Certifique-se de distinguir $\lg^{(i)} n$ (a função logaritmo aplicada i vezes em sucessão, começando com o argumento n) de $\lg^i n$ (o logaritmo de n elevado à i-ésima potência). Então, definimos a função logaritmo iterado como

$$\lg^* n = \min \{i \geq 0 : \lg^{(i)} n \leq 1\}.$$

O logaritmo iterado é uma função que cresce *muito* lentamente:

$$\begin{aligned} \lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^* (2^{65536}) &= 5. \end{aligned}$$

Tendo em vista que o número de átomos no universo visível é estimado em cerca 10^{80}, que é muito menor que 2^{65536}, raramente encontraremos uma entrada de tamanho n tal que $\lg^* n > 5$.

Números de Fibonacci

Definimos os ***números de Fibonacci*** pela seguinte recorrência:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2} \text{ para } i \geq 2. \end{aligned} \tag{3.22}$$

Portanto, cada número de Fibonacci é a soma dos dois números anteriores, produzindo a sequência

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Os números de Fibonacci estão relacionados com a ***razão áurea*** φ e com seu conjugado $\hat{\varphi}$, que são as duas raízes da equação

$$x^2 = x + 1 \tag{3.23}$$

e são dados pelas seguintes fórmulas (veja Exercício 3.2-6):

$$\begin{aligned}\phi &= \frac{1+\sqrt{5}}{2} \qquad (3.24)\\ &= 1{,}61803....\\ \hat{\phi} &= \frac{1-\sqrt{5}}{2}\\ &= -0{,}61803....\end{aligned}$$

Especificamente, temos

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}},$$

que podemos provar por indução (Exercício 3.2-6). Como $|\hat{\varphi}| < 1$, temos

$$\begin{aligned}\frac{\left|\hat{\phi}^i\right|}{\sqrt{5}} &< \frac{1}{\sqrt{5}}\\ &< \frac{1}{2},\end{aligned}$$

o que implica que

$$F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor, \qquad (3.25)$$

o que equivale a dizer que o i-ésimo número de Fibonacci $F\varphi^i$ é igual a / $\sqrt{5}$, arredondado para o inteiro mais próximo. Portanto, os números de Fibonacci crescem exponencialmente.

Exercícios

3.2-1 Mostre que, se $f(n)$ e $g(n)$ são funções monotonicamente crescentes, então as funções $f(n) + g(n)$ e $f(g(n))$ também são e, se além disso, $f(n)$ e $g(n)$ são não negativas, então $f(n) \cdot g(n)$ é monotonicamente crescente.

3.2-2 Prove a equação (3.16).

3.2-3 Prove a equação (3.19). Prove também que $n! = \omega(2_n)$ e que $n! = o(n_n)$.

3.2-4 A função $\lg n!$ é polinomialmente limitada? A função $\lg \lg n!$ é polinomialmente limitada?

3.2-5 Qual é assintoticamente maior: $\lg(\lg^* n)$ ou $\lg^* (\lg n)$?

3.2-6 Prove que a razão áurea φ e seu conjugado $\hat{\varphi}$ satisfazem a equação $x_2 = x + 1$.

3.2-7 Prove por indução que o i-ésimo número de Fibonacci satisfaz à igualdade

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}},$$

onde φ é a razão áurea e $\hat{\varphi}$ é seu conjugado.

3.2-8 Prove que $k \ln k = \Theta(n)$ implica $k = \Theta(n/\ln n)$.

Problemas

3-1 ***Comportamento assintótico de polinômios***

Seja

$$p(n) = \sum_{i=0}^{d} a_i n^i,$$

onde $a_d > 0$, um polinômio de grau d em n, e seja k uma constante. Use as definições das notações assintóticas para provar as propriedades a seguir.

a. Se $k \geq d$, então $p(n) = O(n^k)$.

b. Se $k \leq d$, então $p(n) = \Omega(n^k)$.

c. Se $k = d$, então $p(n) = \Theta(n^k)$.

d. Se $k > d$, então $p(n) = o(n^k)$.

e. Se $k < d$, então $p(n) = (n^k)$.

3-2 ***Crescimentos assintóticos relativos***

Indique, para cada par de expressões (A, B) na tabela a seguir, se A é O, o, Ω, ω ou Θ de B. Considere que $k \geq 1$, $\varepsilon > 0$ e $c > 1$ são constantes. Sua resposta deve estar na forma da tabela, com “sim” ou “não” escrito em cada retângulo.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{sen\, n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

3-3 Ordenação por taxas de crescimento assintóticas

a. Classifique as funções a seguir por ordem de crescimento; isto é, determine um arranjo $g_1, g_2, \ldots, g_{30}$ das funções que satisfazem a $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, ..., $g_{29} = \Omega(g_{30})$. Subdivida sua lista em classes de equivalência tais que $f(n)$ e $g(n)$ estejam na mesma classe se e somente se $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

b. Dê um exemplo de uma função não negativa $f(n)$ tal que, para todas as funções $g_i(n)$ da parte (a), $f(n)$ não seja nem $O(g_i(n))$, nem $\Omega(g_i(n))$.

3-4 Propriedades da notação assintótica

Sejam $f(n)$ e $g(n)$ funções assintoticamente positivas. Prove ou desprove cada uma das seguintes conjeturas.

a. $f(n) = O(g(n))$ implica $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n))$ implica $\lg(f(n)) = O(\lg(g(n)))$, onde $\lg(g(n)) \geq 1$ e $f(n) \geq 1$ para todo n suficientemente grande.

d. $f(n) = O(g(n))$ implica $2^{f(n)} = O(2^{g(n)})$.

e. $f(n) = O((f(n))^2)$.

f. $f(n) = O(g(n))$ implica $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

3-5 Variações para O e Ω

∞

Alguns autores definem Ω de modo ligeiramente diferente de nós; vamos usar $\overset{\infty}{\Omega}$ (lê-se “ômega infinito”) para essa definição alternativa. Dizemos que $f(n) = \overset{\infty}{\Omega}(g(n))$ se existe uma constante positiva c tal que $f(n) \geq cg(n) \geq 0$ para infinitos inteiros n.

a. Mostre que, para quaisquer duas funções $f(n)$ e $g(n)$ que são assintoticamente não negativas, $f(n) = O(g(n))$ ou $f(n) = \overset{\infty}{\Omega}(g(n))$ ou ambas, enquanto isso não é verdade se usarmos Ω em vez de $\overset{\infty}{\Omega}$.

b. Descreva as vantagens e as desvantagens potenciais de se usar $\overset{\infty}{\Omega}$ em vez de Ω para caracterizar os tempos de execução de programas.

Alguns autores também definem O de um modo ligeiramente diferente; vamos usar O' para a definição alternativa. Dizemos que $f(n) = O'(g(n))$ se e somente se $|f(n)| = O(g(n))$.

c. O que acontece para cada direção de “se e somente se” no Teorema 3.1 se substituirmos O por O', mas ainda usarmos Ω?

Alguns autores definem $\tilde{O}$ (lê-se “‘ó suave”) para indicar O com fatores logarítmicos ignorados:

$$\tilde{O}(g(n)) = \{f(n)\text{: existem constantes positivas } c, k \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \lg^k(n) \text{ para todo } n \geq n_0\}\ .$$

d. Defina Ω e Θ de maneira semelhante. Prove a analogia correspondente ao Teorema 3.1.

3-6 Funções iteradas

Podemos aplicar o operador de iteração * usado na função lg* a qualquer função monotonicamente crescente $f(n)$ no domínio dos números reais. Para uma dada constante $c \in$, definimos a função iterada (ou repetida) por

$$f_c^*(n) = \min\{i \geq 0 : f^{(i)}(n) \leq c\}\ ,$$

que não necessita ser bem definida em todos os casos. Em outras palavras, a quantidade $f_c^*(n)$ é o número de aplicações iteradas da função f necessárias para reduzir seu argumento a c ou menos.

Para cada uma das funções $f(n)$ e constantes c a seguir, forneça um limite tão justo quanto possível para $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

NOTAS DO CAPÍTULO

Knuth [182] traça a origem da notação O em texto de teoria dos números escrito por P. Bachmann em 1892. A notação o foi criada por E. Landau, em 1909, para sua discussão da distribuição de números primos. As notações Ω e Θ foram defendidas por Knuth [213] para corrigir a prática popular, mas tecnicamente descuidada, de se usar na literatura a notação O para os limites superiores e inferiores. Muitas pessoas continuam a usar a notação O onde a notação Θ é mais precisa tecnicamente. Uma discussão adicional da história e do desenvolvimento de notações assintóticas pode ser encontrada em obras de Knuth [209, 213] e em Brassard e Bratley [54]. Nem todos os autores definem as notações assintóticas do mesmo modo, embora as várias definições concordem na maioria das situações comuns. Algumas das definições alternativas abrangem funções que não são assintoticamente não negativas, desde que seus valores absolutos sejam adequadamente limitados.

A equação (3.20) se deve a Robbins [297]. Outras propriedades de funções matemáticas elementares podem ser encontradas em qualquer bom livro de referência de matemática, como Abramowitz e Stegun [1] ou Zwillinger [362], ou em um livro de cálculo, como Apostol [18] ou Thomas et al. [334]. Knuth [209] e Graham, Knuth e Patashnik [152] contêm grande quantidade de material sobre matemática discreta, tal como utilizada em ciência da computação.

---

1 Na notação de conjuntos, um sinal de dois-pontos deve ser lido como “tal que”.

2 O problema real é que nossa notação comum para funções não distingue entre funções e valores. Em cálculo , os parâmetros para uma função são claramente especificados: a função n^2 poderia ser escrita como $n.n^2$ ou até mesmo $r.r^2$. Porém, a adoção de uma notação mais rigorosa complicaria manipulações algébricas e, assim, optamos por tolerar o abuso.

4 DIVISÃO E CONQUISTA

Na Seção 2.3.1, vimos como a ordenação por intercalação serve como exemplo do paradigma divisão e conquista. Lembre-se de que, segundo esse paradigma, resolvemos um problema recursivamente aplicando três **etapas** em cada nível da recursão:

Divisão o problema em certo número de subproblemas que são instâncias menores do mesmo problema.

Conquista os subproblemas resolvendo-os recursivamente. Entretanto, se os tamanhos dos subproblemas forem suficientemente pequenos, basta resolvê-los de modo direto.

Combinação as soluções dos subproblemas na solução para o problema original.

Quando os subproblemas são suficientemente grandes para serem resolvidos recursivamente, trata-se de um ***caso recursivo***. Logo que os subproblemas se tornam suficientemente pequenos que não mais recorremos à recursão, dizemos que a recursão "se esgotou" e que chegamos ao ***caso-base***. Às vezes, além de subproblemas que são instâncias menores do mesmo problema, temos de resolver subproblemas que não são exatamente iguais ao problema original. Consideramos a solução de tais problemas como parte da etapa "combinar".

Neste capítulo, veremos mais algoritmos baseados em divisão e conquista. O primeiro resolve o problema do subarranjo máximo: toma como entrada um arranjo de números e determina o subarranjo contíguo cujos valores resultem na maior soma.

Então, estudaremos dois algoritmos de divisão e conquista para multiplicar matrizes $n \times n$. O tempo de execução de um deles é $\Theta(n_3)$, o que não é melhor do que o método direto da multiplicação de matrizes quadradas. Porém, o tempo de execução do outro, o algoritmo de Strassen, é $O(n_{2.81})$, que supera assintoticamente o método direto.

Recorrências

As recorrências andam de mãos dadas com o paradigma divisão e conquista, porque nos dão um modo natural de caracterizar os tempos de execução de algoritmos de divisão e conquista. Uma ***recorrência*** é uma equação ou desigualdade que descreve uma função em termos de seu valor para entradas menores. Por exemplo, na Seção 2.3.2 descrevemos o tempo de execução do pior caso $T(n)$ do procedimento MERGE-SORT pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases} \tag{4.1}$$

cuja solução afirmamos ser $T(n) = \Theta(n \lg n)$.

As recorrências podem tomar muitas formas. Por exemplo, um algoritmo recursivo poderia dividir problemas em tamanhos desiguais, como uma subdivisão 2/3 para 1/3. Se as etapas de divisão e combinação levarem tempo linear, tal algoritmo dará origem à recorrência $T(n) = T(2n/3) + T(n/3) + \Theta(n)$.

Os subproblemas não estão necessariamente restritos a ser uma fração constante do tamanho do problema original. Por exemplo, uma versão recursiva da busca linear (veja Exercício 2.1-3) criaria apenas um subproblema contendo

somente um elemento a menos do que o problema original. Cada chamada recursiva levaria tempo constante mais o tempo das chamadas recursivas que fizer, o que produz a recorrência $T(n) = T(n - 1) + \Theta(1)$.

Este capítulo apresenta três métodos para resolver recorrências, isto é, para obter limites assintóticos "Θ" ou "O" para a solução.

- No ***método de substituição***, arriscamos um palpite para um limite e então usamos indução matemática para provar que nosso palpite estava correto.
- O ***método da árvore de recursão*** converte a recorrência em uma árvore cujos nós representam os custos envolvidos em vários níveis da recursão. Usamos técnicas para limitar somatórios para resolver a recorrência.
- O ***método mestre*** dá limites para recorrências da forma

$$T(n) = aT(n/b) + f(n), \tag{4.2}$$

onde $a \geq 1$, $b > 1$ e $f(n)$ é uma função dada. Tais recorrências ocorrem frequentemente. Uma recorrência da forma da equação (4.2) caracteriza um algoritmo de divisão e conquista que cria a subproblemas, cada um com 1/b do tamanho do problema original e no qual as etapas de divisão e conquista, juntas, levam o tempo $f(n)$.

Para utilizar o método mestre, você terá de memorizar três casos; porém, com isso, será fácil determinar limites assintóticos para muitas recorrências simples. Usaremos o método mestre para determinar os tempos de execução de algoritmos de divisão e conquista para o problema do subarranjo máximo e para a multiplicação de matrizes, bem como para outros algoritmos baseados no método de divisão e conquista em outros lugares neste livro.

Ocasionalmente veremos recorrências que não são igualdades, porém, mais exatamente, desigualdades, como $T(n) \leq 2T(n/2) + \Theta(n)$. Como tal recorrência declara somente um limite superior para $T(n)$, expressaremos sua solução usando a notação O em vez da notação Θ. De modo semelhante, se a desigualdade for invertida para $T(n) \geq 2T(n/2) + \Theta(n)$, então, como a recorrência dá apenas um limite inferior para $T(n)$, usaríamos a notação Ω em sua solução.

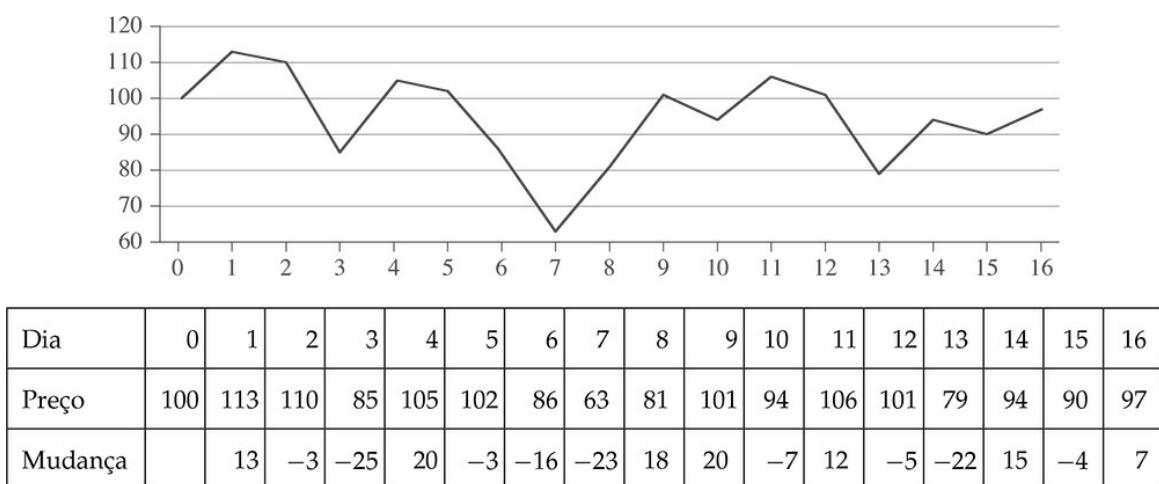
Detalhes técnicos

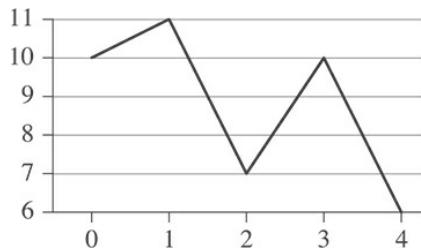
Na prática, negligenciamos certos detalhes técnicos quando enunciamos e resolvemos recorrências. Por exemplo, se chamarmos MERGE-SORT para n elementos quando n é ímpar, terminaremos com subproblemas de tamanho $n/2$ e $n/2$. Nenhum dos tamanhos é realmente $n/2$ porque $n/2$ não é um inteiro quando n é ímpar. Tecnicamente, a recorrência que descreve o tempo de execução de MERGE-SORT é, na realidade,

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n > 1. \end{cases} \tag{4.3}$$

As condições de contorno representam uma outra classe de detalhes que em geral ignoramos. Visto que o tempo de execução de um algoritmo para uma entrada de tamanho constante é uma constante, as recorrências que surgem dos tempos de execução de algoritmos geralmente têm $T(n) = \Theta(1)$ para n suficientemente pequeno. Em consequência disso, por conveniência, em geral omitiremos declarações sobre as condições de contorno de recorrências e consideraremos que $T(n)$ é constante para n pequeno. Por exemplo, normalmente enunciamos a recorrência (4.1) como

$$T(n) = 2T(n/2) + \Theta(n) \tag{4.4}$$

sem atribuir explicitamente valores para n pequeno. A razão é que, embora mudar o valor de $T(1)$ altere a solução exata para a recorrência, normalmente a solução não muda por mais de um fator constante, e assim a ordem de crescimento não é alterada.

Quando enunciamos e resolvemos recorrências, muitas vezes, omitimos pisos, tetos e condições de contorno. Seguimos em frente sem esses detalhes, e mais tarde determinamos se eles têm importância ou não. Em geral não têm, mas é importante saber quando têm. A experiência ajuda, e também alguns teoremas que afirmam que esses detalhes

não afetam os limites assintóticos de muitas recorrências que caracterizam os algoritmos de divisão e conquista (veja o Teorema 4.1). Porém, neste capítulo, examinaremos alguns desses detalhes e ilustraremos os bons aspectos de métodos de solução de recorrência.

4.1 O PROBLEMA DO SUBARRANJO MÁXIMO

Suponha que lhe tenha sido oferecida a oportunidade de investir na Volatile Chemical Corporation. Assim como os produtos químicos que a empresa produz, o preço da ação da Volatile Chemical Corporation também é bastante volátil. Você só pode comprar uma única unidade de ação somente uma vez e então vendê-la em data posterior. Além disso, as operações de compra e venda só podem ser executadas após o fechamento do pregão do dia. Para compensar essa restrição, você pode saber qual será o preço da ação no futuro. Sua meta é maximizar seu lucro. A Figura 4.1 mostra o preço da ação durante um período de 17 dias. Você pode comprar a ação a qualquer tempo, começando depois do dia 0, quando o preço é $100 por ação. Claro que você gostaria de "comprar na baixa e vender na alta" — comprar ao preço mais baixo possível e mais tarde vender ao preço mais alto possível — para maximizar seu lucro. Infelizmente, pode ser que você não consiga comprar ao preço mais baixo e vender ao preço mais alto dentro de um determinado período. Na Figura 4.1, o preço mais baixo ocorre depois do dia 7, que ocorre após o preço mais alto, depois do dia 1.

Você poderia achar que sempre pode maximizar o lucro comprando ao preço mais baixo ou vendendo ao preço mais alto. Por exemplo, na Figura 4.1, maximizaríamos o lucro comprando ao preço mais baixo após o dia 7. Se essa estratégia sempre funcionasse, seria fácil determinar como maximizar o lucro: localizar o preço mais alto de todos e o preço mais baixo de todos, e então percorrer os dados começando na esquerda a partir do preço mais alto para achar o preço mais baixo anterior, e examinar os dados começando na direita a partir do preço mais baixo para achar o último preço mais alto e escolher o par que apresentasse a maior diferença. A Figura 4.2 apresenta um contraexemplo simples que demonstra que, às vezes, o lucro máximo não ocorre quando compramos ao preço mais baixo nem quando vendemos ao preço mais alto.

Dia	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Preço	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Mudança		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figura 4.1 Informações sobre o preço da ação da Volatile Chemical Corporation ao final do pregão diário durante um período de 17 dias. O eixo horizontal do diagrama indica o dia, e o eixo vertical mostra o preço. A linha inferior da tabela dá a mudança no preço ocorrida no dia anterior.

Dia	0	1	2	3	4
Preço	10	11	7	10	6
Mudança		1	–4	3	–4

Figura 4.2 Um exemplo que mostra que o lucro máximo nem sempre começa no preço mais baixo ou termina no preço mais alto. Novamente, o eixo horizontal indica o dia, e o eixo vertical mostra o preço. Aqui, o lucro máximo de $3 por ação seria auferido comprando após o dia 2 e vendendo após o dia 3. No global, o preço de $7 após o dia 2 não é o mais baixo e o preço de $10 após o dia 3 não é o mais alto.

Uma solução de força bruta

Podemos propor facilmente uma solução de força bruta para esse problema: experimente todo par possível de datas de compra e venda no qual a data de compra seja anterior à data de venda. Um período de n dias tem $\binom{n}{2}$ de tais pares de dados. Visto que $\binom{n}{2}$ + é $\Theta(n^2)$ e o melhor que podemos esperar é avaliar cada par de dados em tempo constante, essa abordagem levaria um tempo de $\Omega(n_2)$. Poderíamos conseguir algo melhor?

Uma transformação

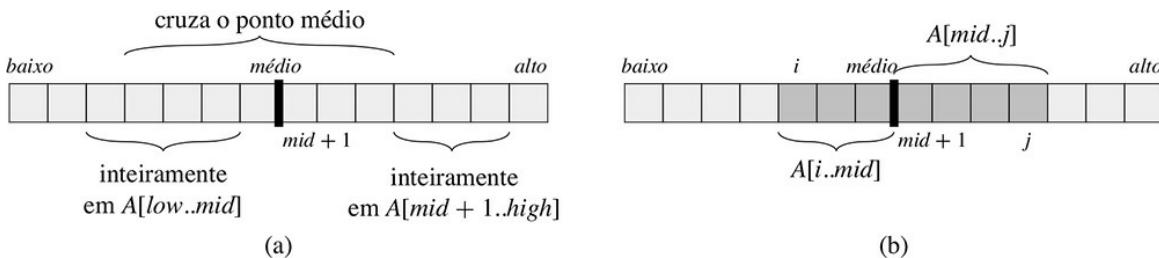
Para projetar um algoritmo com tempo de execução $o(n_2)$, vamos considerar os dados de entrada de um modo ligeiramente diferente. Queremos determinar uma sequência de dias durante a qual a mudança líquida desde o primeiro dia até o último é máxima. Em vez de examinar os preços diários, vamos considerar a alteração diária nos preços, sendo que a mudança no dia i é a diferença entre os preços após o dia i - 1 e após o dia i. A tabela na Figura 4.1 mostra essas mudanças diárias na última linha. Se tratarmos essa linha como um arranjo A, mostrado na Figura 4.3, vamos querer determinar o subarranjo não vazio contíguo a A cujos valores tenham a maior soma. Denominamos esse subarranjo contíguo ***subarranjo máximo***. Por exemplo, no arranjo da Figura 4.3, o subarranjo máximo de A[1 . . 16] é A[8 . . 11], com soma 43. Assim, o mais interessante seria você comprar a ação imediatamente antes do dia 8 (isto é, após o dia 7) e vendê-la depois do dia 11, auferindo um lucro de $43 por ação.

À primeira vista, essa transformação não ajuda. Ainda precisamos verificar $\binom{n-1}{2} = \Theta(n2)$ subarranjos para um período de n dias. O Exercício 4.1-2 pede que você mostre que, embora o cálculo de um único subarranjo possa levar tempo proporcional ao comprimento do subarranjo, quando calculamos todas as $\Theta(n_2)$ somas de subarranjos, podemos organizar o cálculo de modo que cada soma de subarranjo leve o tempo $O(1)$, dados os valores de somas de subarranjos calculados anteriormente, de modo que a solução de força bruta leva o tempo $\Theta(n_2)$.

Figura 4.3 A mudança nos preços da ação como um problema de subarranjo máximo. Aqui, o subarranjo A[8 . . 11], com soma 43, tem a maior soma de qualquer subarranjo contíguo do arranjo A.

Portanto, vamos procurar uma solução mais eficiente para o problema do subarranjo máximo. Nesse processo, normalmente falaremos de “um” subarranjo máximo em vez de “o” subarranjo máximo, já que poderia haver mais de um

subarranjo que alcance a soma máxima.

O problema do subarranjo máximo é interessante somente quando o arranjo contém alguns números negativos. Se todas as entradas do arranjo fossem não negativas, o problema do subarranjo máximo não representaria nenhum desafio, já que o arranjo inteiro daria a maior soma.

Uma solução utilizando divisão e conquista

Vamos pensar em como poderíamos resolver o problema do subarranjo máximo usando a técnica de divisão e conquista. Suponha que queiramos determinar um subarranjo máximo do subarranjo A[*low* . . *high*]. O método de divisão e conquista sugere que dividamos o subarranjo em dois subarranjos, com tamanhos mais iguais dentro do possível. Isto é, determinamos o ponto médio, digamos *mid*, do subarranjo, e consideramos os subarranjos A[*low* . . *mid*] e A[*mid* + 1 . . *high*]. Como mostra a Figura 4.4(a), qualquer subarranjo contíguo A[i . . j] de A[*low* . . *high*] deve encontrar-se exatamente em um dos seguintes lugares:

- inteiramente no subarranjo A[*low* . . *mid]*, de modo que $low \le i \le j \le mid$,
- inteiramente no subarranjo A[*mid* + 1 . . *high]*, de modo que $mid < i \le j \le high$, ou
- cruzando o ponto médio, de modo que $low \le i \le mid < j \le high$.

Portanto, um subarranjo máximo de A[*low* . . *high]* deve encontrar-se exatamente em um desses lugares. Na verdade, um subarranjo máximo de A[*low* . . *high]* deve ter a maior soma de todos os subarranjos inteiramente em A[*low* . . *mid*], inteiramente em A[*mid* + 1 . . *high*] ou cruzando o ponto médio. Podemos determinar subarranjos máximos de A[*low* . . *mid*] e A[*mid*+1 . . *high*] recursivamente porque esses dois subproblemas são instâncias menores do problema da determinação de um subarranjo máximo. Assim, resta apenas encontrar um subarranjo máximo que cruze o ponto médio e tomar um subarranjo que tenha a maior soma dos três.

Podemos encontrar facilmente um subarranjo máximo que cruze o ponto médio em tempo linear do tamanho do subarranjo *A*[*low* . . *high]*. Esse problema *não é* uma instância menor de nosso problema original, porque há a restrição adicional de que o subarranjo que ele escolher deve cruzar o ponto médio. Como mostra a Figura 4.4(b), qualquer subarranjo que cruze o ponto médio é composto por dois subarranjos *A*[*i* . . *mid]* e *A*[*mid* + 1 . . *j*], onde $low \le i \le mid$ *e* $mid < j \le high$. Portanto, precisamos apenas encontrar subarranjos máximos da forma *A*[i . . *mid]* e *A*[*mid* +1 . . j] e então combiná-los. O procedimento FIND-MAX-CROSSING-SUBARRAY toma como entrada o arranjo *A* e os índices *low*, *mid e high*, e retorna uma tupla que contém os índices que demarcam um subarranjo máximo que cruza o ponto médio, juntamente com a soma dos valores em um subarranjo máximo.

Figura 4.4 (a) Possíveis localizações de subarranjos de A[*low* . . *high*]: inteiramente em A[*low* . . *mid*], inteiramente em A[*mid* + 1 . . *high*] ou cruzando o ponto médio *mid*. **(b)** Qualquer subarranjo de A[*low* . . *high*] que cruze o ponto médio compreende dois subarranjos A[*i* . . *mid*] e A[*mid* + 1 . . *j*], onde $low \le i \le mid$ e $mid < j \le high$.

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
1 left-sum = −∞
2 sum = 0
3 for i = mid downto low
4         sum = sum + A[i]
5         if sum > left-sum
6                 left-sum = sum
7                 max-left = i
8 right-sum = −∞
9 sum = 0
10 for j = mid + 1 to high
11        sum = sum + A[j]
12        if sum > right-sum
13                right-sum = sum
14                max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```

Esse procedimento funciona da seguinte maneira: as linhas 1-7 acham um subarranjo máximo da metade esquerda, $A[low\ ..\ mid]$. Visto que esse subarranjo deve conter A[*mid*], o laço **for** das linhas 3-7 inicia o índice *i* em *mid* e prossegue até *low*, de modo que todo subarranjo que ele considera é da forma $A[i\ ..\ mid]$. As linhas 1-2 inicializam as variáveis *left-sum*, que contêm a maior soma encontrada até então, e *sum*, que contém as somas das entradas em $A[i\ ..\ mid]$. Sempre que encontrarmos, na linha 5, um subarranjo $A[i\ ..\ mid]$ com uma soma de valores maior do que *left-sum*, atualizaremos *left-sum para a soma desse* subarranjo na linha 6, e na linha 7 atualizaremos a variável *max-left* para registrar esse índice *i*. As linhas 8-14 funcionam de modo análogo para a metade direita, $A[mid+1\ ..\ high]$. Aqui, o laço **for** das linhas 10-14 inicia o índice *j* em *mid*+1 e prossegue até *high*, de modo que todo subarranjo que ele considera é da forma $A[mid + 1\ ..\ j\ ]$. Finalmente, a linha 15 retorna os índices *max-left* e *max-right* que demarcam um subarranjo máximo que cruza o ponto médio, juntamente com a soma *left-sum*+*right-sum* dos valores no subarranjo $A[max\text{-}left\ ..\ max\text{-}right]$.

Se o subarranjo $A[low\ ..\ high]$ contiver n entradas (de modo que $n = high - low + 1$), afirmamos que a chamada FIND-MAX-CROSSING-SUBARRAY(*A*, *low*, *mid*, *high*) leva o tempo $\Theta(n)$. Visto que cada iteração de cada um dos dois laços **for** leva o tempo $\Theta(1)$, precisamos apenas contar quantas iterações há no total. O laço **for das** linhas 3-7 faz $mid - low + 1$ iterações e o laço **for** das linhas 10-14 faz $high - mid$ iterações e, assim, o número total de iterações é

$$\begin{aligned}(mid - low + 1) + (high - mid) &= high - low + 1 \\ &= n\ .\end{aligned}$$

Com um procedimento FIND-MAX-CROSSING-SUBARRAY de tempo linear à mão, podemos escrever pseudocódigo para um algoritmo de divisão e conquista para resolver o problema do arranjo máximo:

```
FIND-MAX-CROSSING-SUBARRAY(A; low; high)
1 if high == low
2       return (low; high;A[low])                    // caso base: só um elemento
3 else mid = ⌊(low + high)/2⌋
4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBADRRAY(A, low, mid)
5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7       if left-sum ≥ right-sum e left-sum ≥ cross-sum
8             return (left-low, left-high, left-sum)
9       elseif right-sum ≥ left-sum e right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

A chamada inicial FIND-MAXIMUM-SUBARRAY$(A, 1, A.\ length)$ encontrará um subarranjo máximo de $A[1\ .\ .\ n]$.

Assim como FIND-MAX-CROSSING-SUBARRAY, o procedimento recursivo FIND-MAXIMUM-SUBARRAY retorna uma tupla que contém os índices que demarcam um subarranjo máximo, juntamente com a soma dos valores em um subarranjo máximo. A linha 1 testa o caso-base, no qual o subarranjo tem apenas um elemento. Um subarranjo que tenha apenas um elemento tem somente um subarranjo — ele mesmo — e, assim, a linha 2 retorna uma tupla com os índices de início e fim daquele único elemento, juntamente com seu valor. As linhas 3-11 tratam o caso recursivo. A linha 3 executa a parte da divisão, calculando o índice *mid* do ponto médio. Vamos nos referir ao subarranjo $A[low\ .\ .\ mid]$ como o ***subarranjo da esquerda*** e o subarranjo $A[mid + 1\ .\ .\ high]$ como o ***subarranjo da direita***. Como sabemos que o subarranjo $A[low\ .\ .\ high]$ contém no mínimo dois elementos, cada um dos subarranjos, o da direita e o da esquerda, deve ter no mínimo um elemento. As linhas 4 e 5 conquistam por encontrarem recursivamente subarranjos máximos dentro dos subarranjos da esquerda e da direita, respectivamente.

As linhas 6-11 formam a parte de combinar. A linha 6 encontra um subarranjo máximo que cruza o ponto médio. (Lembre-se de que, como a linha 6 resolve um subproblema que não é uma instância menor do problema original, consideramos que ela está na parte de combinar.) A linha 7 testa se o subarranjo da esquerda contém um subarranjo que tenha a soma máxima, e a linha 8 retorna esse subarranjo máximo. Senão, a linha 9 testa se o subarranjo da direita contém um subarranjo que tenha a soma máxima, e a linha 10 retorna esse subarranjo máximo. Se nenhum dos subarranjos, o da direita ou o da esquerda, contiver um subarranjo que atinja a soma máxima, então um subarranjo máximo deve cruzar o ponto médio, e a linha 11 o retorna.

Análise do algoritmo de divisão e conquista

A seguir, montaremos uma recorrência que descreve o tempo de funcionamento do procedimento recursivo FIND MAXIMUM-SUBARRAY. Assim como fizemos na Seção 2.3.2, quando analisamos a ordenação por intercalação, adotamos aqui a premissa simplificadora de que o tamanho do problema original é uma potência de 2, de modo que os tamanhos de todos os subproblemas são números inteiros. Denotamos por $T(n)$ o tempo de execução de FIND-MAXIMUM-SUBARRAY para um subarranjo de n elementos. Para começar, a linha 1 adota tempo constante. O caso-base, quando $n = 1$, é fácil: a linha 2 leva tempo constante e, assim,

$$T(1) = \Theta(1) \tag{4.5}$$

O caso recursivo ocorre quando $n > 1$. As linhas 1 e 3 levam tempo constante. Cada um dos subproblemas resolvidos nas linhas 4 e 5 trata de um subarranjo de $n/2$ elementos (visto que a premissa adotada é que o tamanho original do problema é uma potência de 2, fica garantido que $n/2$ é um inteiro) e por isso, despendemos o tempo $T(n/2)$

para resolver cada um deles. Como temos de resolver dois subproblemas — para o subarranjo da esquerda e para o subarranjo da direita —, a contribuição para o tempo de execução dada pelas linhas 4 e 5 chega a $2T(n/2)$. Como já vimos, a chamada a FIND-MAX-CROSSING-SUBARRAY na linha 6 leva o tempo $\Theta(n)$. As linhas 7-11 levam somente o tempo $\Theta(1)$. Portanto, para o caso recursivo, temos

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) \end{aligned} \tag{4.6}$$

Combinando as equações (4.5) e (4.6), obtemos uma recorrência para o tempo de execução $T(n)$ de FIND-MAX-CROSSING-SUBARRAY:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1, \\ 2T(n/2) + \Theta(n) & \text{se } n > 1. \end{cases} \tag{4.7}$$

Essa recorrência é igual à recorrência (4.1) para a ordenação por intercalação. Como veremos pelo método mestre na Seção 4.5, a solução dessa recorrência é $T(n) = \Theta(n \lg n)$. Também seria bom você rever a árvore de recursão na Figura 2.5 para entender por que a solução deve ser $T(n) = \Theta(n \lg n)$.

Assim, vemos que o método de divisão e conquista produz um algoritmo que é assintoticamente mais rápido do que o método da força bruta. Com a ordenação por intercalação, e agora o problema do subarranjo máximo, começamos a ter uma ideia do poder do método de divisão e conquista. Às vezes, ele produzirá o algoritmo assintoticamente mais rápido para um problema, e outras vezes podemos nos sair ainda melhor. Como mostra o Exercício 4.1-5, na verdade existe um algoritmo de tempo linear para o problema do subarranjo máximo, e esse algoritmo não usa o método de divisão e conquista.

Exercícios

4.1-1 O que FIND-MAXIMUM-SUBARRAY retorna quando todos os elementos de A são negativos?

4.1-2 Escreva pseudocódigo para o método da força bruta de solução do problema do subarranjo máximo. Seu procedimento deve ser executado no tempo $\Theta(n_2)$.

4.1-3 Implemente em seu computador o algoritmo da força bruta e também o algoritmo recursivo para o problema do subarranjo máximo. Qual é o tamanho de problema n_0 que dá o ponto de cruzamento no qual o algoritmo recursivo supera o algoritmo da força bruta? Em seguida, mude o caso-base do algoritmo recursivo para usar o algoritmo de força bruta sempre que o tamanho do problema for menor que n_0. Isso muda o ponto de cruzamento?

4.1-4 Suponha que mudamos a definição do problema do subarranjo máximo para permitir que o resultado seja um subarranjo vazio, no qual a soma dos valores de um subarranjo vazio é 0. Como você mudaria qualquer dos algoritmos que não aceitam subarranjos vazios de modo a permitir que o resultado seja um subarranjo vazio?

4.1-5 Use as seguintes ideias para desenvolver um algoritmo não recursivo de tempo linear para o problema do subarranjo máximo. Comece na extremidade esquerda do arranjo e prossiga em direção à extremidade direita, sem perder de vista o subarranjo máximo visto até aqui. Conhecendo um subarranjo máximo de $A[1 \mathinner{..} j]$, estenda a resposta para achar um subarranjo máximo que termine no índice $j + 1$ usando a seguinte observação: um subarranjo máximo de $A[1 \mathinner{..} j + 1]$ é um subarranjo máximo de $A[1 \mathinner{..} j]$ ou um subarranjo $A[i \mathinner{..} j + 1]$, para algum $1 \leq i \leq j + 1$. Determine um subarranjo máximo da forma $A[i \mathinner{..} j + 1]$ em tempo constante com base no conhecimento de um subarranjo máximo que termine no índice j.

4.2 ALGORITMO DE STRASSEN PARA MULTIPLICAÇÃO DE MATRIZES

Se você já viu matrizes antes, provavelmente sabe como multiplicá-las. (Caso contrário, deve ler a Seção D.1 no Apêndice D.) Se $A = (a_{ij})$ e $B = (b_{ij})$ são matrizes quadradas $n \times n$, então no produto $C = A \times B$, definimos a entrada c_{ij} , para $i, j = 1, 2, \ldots , n$, por

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \ . \tag{4.8}$$

Temos de calcular n_2 entradas de matrizes, e cada uma é a soma de n valores. O procedimento descrito a seguir toma as matrizes $n \times n$ A e B e as multiplica, retornando seu produto C. Consideramos que cada matriz tem um atributo *rows* (*linhas*), *que dá o número de linhas na* matriz.

SQUARE-MATRIX-MULTIPLY(A,B)
1 $n = A{\cdot}rows$
2 seja C uma nova matriz $n \times n$
3 **for** $i = 1$ **to** n
4   **for** $j = 1$ **to** n
5         $c_{ij} = 0$
6   **for** $k = 1$ **to** n
7             $c_{ij} = c_{ij} = a_{ik} \cdot b_{kj}$
8 **return** C

O procedimento SQUARE-MATRIX-MULTIPLY funciona da seguinte maneira: o laço **for** das linhas 3-7 calcula as entradas de cada linha i e, dentro de uma linha i dada, o laço **for** das linhas 4-7 calcula cada uma das entradas c_{ij}, para cada coluna j. A linha 5 inicializa c_{ij} em 0 quando começamos a calcular a soma dada na equação (4.8), e cada iteração do laço **for** das linhas 6-7 acrescenta mais um termo da equação (4.8).

Como cada um dos laços **for** triplamente aninhados executa exatamente n iterações, e cada execução da linha 7 leva tempo constante, o procedimento SQUARE-MATRIX-MULTIPLY leva o tempo $\Theta(n_3)$.

É possível que a princípio você pense que qualquer algoritmo de multiplicação de matrizes tem de levar o tempo (n_3), já que a definição natural de multiplicação de matrizes requer aquele mesmo tanto de multiplicações. Porém, você estaria errado: temos um modo de multiplicar matrizes no tempo $o(n_3)$. Nesta seção, veremos o notável algoritmo recursivo de Strassen para multiplicar matrizes $n \times n$. O tempo de execução desse algoritmo é $\Theta(n_{\lg}\ 7)$, que demonstraremos na Seção 4.5. Visto que lg 7 encontra-se entre 2,80 e 2,81, o algoritmo de Strassen é executado no tempo $O(n_2,{}^{81})$, que é assintoticamente melhor do que o procedimento simples SQUARE-MATRIX-MULTIPLY.

Um algoritmo simples de divisão e conquista

Por questão de simplicidade, quando usamos um algoritmo de divisão e conquista para calcular o produto de matrizes $C = A \cdot B$, supomos que n é uma potência exata de 2 em cada uma das matrizes $n \times n$. Adotamos essa premissa porque, em cada etapa da divisão, dividiremos matrizes $n \times n$ em quatro matrizes $n/2 \times n/2$ e, considerando que n é uma potência exata de 2, fica garantido que, desde que $n \geq 2$, a dimensão $n/2$ é um número inteiro.

Suponha que repartimos cada uma das matrizes A, B e C em quatro matrizes $n/2 \times n/2$

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

de modo que reescrevemos a equação $C = A \times B$ como

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

A equação (4.10) corresponde às quatro equações

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21},$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22},$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21},$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}.$$

Cada uma dessas quatro equações especifica duas multiplicações de matrizes $n/2 \times n/2$ e a soma de seus produtos $n/2 \times n/2$. Podemos usar essas equações para criar um algoritmo de divisão e conquista recursivo e direto:

```
FIND-MAX-CROSSING-SUBARRAY(A; low; high)
1 if high == low
2       return (low; high;A[low])                    // caso base: só um elemento
3 else mid = ⌊(low + high)/2⌋
4       (left-low, left-high, left-sum) =
                FIND-MAXIMUM-SUBADRRAY(A, low, mid)
5       (right-low, right-high, right-sum) =
                FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6       (cross-low, cross-high, cross-sum) =
                FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7       if left-sum ≥ right-sum e left-sum ≥ cross-sum
8               return (left-low, left-high, left-sum)
9       elseif right-sum ≥ left-sum e right-sum ≥ cross-sum
10              return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

Esse pseudocódigo camufla um detalhe de implementação sutil porém importante. Como repartimos as matrizes na linha 5? Se fôssemos criar 12 novas matrizes $n/2 \times n/2$, gastaríamos o tempo $\Theta(n_2)$ copiando entradas. Na verdade, podemos repartir matrizes sem copiar entradas. O truque é usar cálculos de índices. Identificamos uma submatriz por uma faixa de índices de linha e uma faixa de índices de colunas da matriz original. Terminamos por representar uma submatriz de um modo um pouco diferente daquele que utilizamos para representar a matriz original, que é a sutileza que

estamos camuflando. A vantagem é que, visto que podemos especificar matrizes por cálculo de índices, a execução da linha 5 leva apenas o tempo $\Theta(1)$ (embora veremos que, para o tempo de execução global, não faz diferença assintoticamente se copiamos ou repartimos no lugar).

Agora, derivaremos uma recorrência para caracterizar o tempo de execução de Square-Matrix-Multiply-Recursive. Seja $T(n)$ o tempo para multiplicar duas matrizes $n \times n$ usando esse procedimento. No caso-base, quando $n = 1$, executamos apenas a única multiplicação escalar na linha 4 e, assim,

$$T(1) = \Theta(1) \tag{4.15}$$

O caso recursivo ocorre quando $n > 1$. Como já discutimos, repartir as matrizes na linha 5 leva o tempo $\Theta(1)$ usando cálculo de índices. Nas linhas 6-9, chamamos recursivamente Square-Matrix-Multiply-Recursive um total de oito vezes. Como cada chamada recursiva multiplica duas matrizes $n/2 \times n/2$, o que contribui com $T(n/2)$ para o tempo de execução global, o tempo que leva para todas as oito chamadas recursivas é $8T(n/2)$. Também temos de levar em conta as quatro adições de matrizes nas linhas 6-9. Cada uma dessas matrizes contém $n_2/4$ entradas e, portanto, cada uma das quatro adições de matrizes leva o tempo de $\Theta(n_2)$. Visto que o número de adições de matrizes é uma constante, o tempo total gasto na soma das matrizes nas linhas 6-9 é $\Theta(n_2)$. (Novamente, usamos o cálculo de índices para colocar os resultados das adições de matizes nas posições corretas da matriz C, com um acréscimo de tempo $\Theta(1)$ por entrada.) Portanto, o tempo total para o caso recursivo é a soma do tempo de partição, do tempo para todas as chamadas recursivas e do tempo de adição de matrizes resultantes das chamadas recursivas:

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2). \end{aligned} \tag{4.16}$$

Note que, se implementássemos a partição copiando matrizes, o que custaria o tempo $\Theta(n_2)$, a recorrência não mudaria e, por consequência, o tempo de execução global aumentaria somente por um fator constante.

Combinando as equações (4.15) e (4.16) obtemos a recorrência para o tempo de execução de Square-Matrix-Multiply-Recursive:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1; \\ 8T(n/2) + \Theta(n^2) & \text{se } n > 1. \end{cases} \tag{4.17}$$

Como veremos pelo método mestre na Seção 4.5, a solução da recorrência (4.17) é $T(n) = \Theta(n_3)$. Assim, essa abordagem simples do método de divisão e conquista não é mais rápida do que o procedimento direto Square-Matrix-Multiply.

Antes de passarmos para o exame do algoritmo de Strassen, vamos revisar de onde vieram os componentes da equação (4.16). A partição de cada matriz $n \times n$ por cálculo de índice leva o tempo $\Theta(1)$, mas temos duas matrizes para repartir. Se bem que poderíamos dizer que a partição das duas matrizes leva o tempo $\Theta(2)$, a constante de 2 está incorporada na notação Θ. A soma de duas matrizes, cada uma, digamos, com k entradas, leva o tempo $\Theta(k)$. Visto que cada uma das matrizes que somamos tem $n_2/4$ entradas, poderíamos dizer que a soma de cada par leva o tempo $\Theta(n_2/4)$. Entretanto, novamente a notação Θ incorpora o valor constante de 1/4, e dizemos que somar duas matrizes $n/4 \times n/4$ leva o tempo $\Theta(n_2)$. Temos quatro dessas adições de matrizes e, mais uma vez, em vez de dizermos que elas levam o tempo $\Theta(4n_2)$, dizemos que levam o tempo $\Theta(n_2)$. (Uma observação óbvia é que poderíamos dizer que as quatro adições de matrizes levam o tempo $\Theta(4n_2/4)$ e que $4n_2/4 = n_2$, mas aqui o ponto a ressaltar é que a notação Θ incorpora fatores constantes, sejam quais forem.) Assim, terminamos com dois termos de $\Theta(n_2)$, que podemos combinar em um só.

Todavia, quando levamos em conta as oito chamadas recursivas, não podemos apenas incorporar o fator constante de 8. Em outras palavras, temos de dizer que, juntas, elas levam o tempo $8T(n/2)$, em vez de apenas $T(n/2)$. Podemos ter uma ideia do porquê examinando na árvore de recursão da Figura 2.5 a recorrência (2.1), que é idêntica à recorrência (4.7), com o caso recursivo $\mathrm{T}(n) = 2T(n/2) + \Theta(n)$. O fator de 2 determinou quantos filhos cada nó da

árvore tem, o que, por sua vez, determinou quantos termos contribuíram para a soma em cada nível da árvore. Caso ignorássemos o fator de 8 na equação (4.16) ou o fator de 2 na recorrência (4.1), a árvore de recursão seria apenas linear, em vez de "frondosa", e cada nível contribuiria com apenas um termo para a soma.

Portanto, tenha sempre em mente que, embora a notação assintótica incorpore fatores multiplicativos constantes, o mesmo não acontece com notação recursiva como $T(n/2)$.

O método de Strassen

A chave para o método de Strassen é tornar a árvore de recursão ligeiramente menos frondosa. Isto é, em vez de efetuar oito multiplicações recursivas de $n/2 \times n/2$ matrizes, ele efetua somente sete. O custo de eliminar uma multiplicação de matrizes será várias novas somas de matrizes $n/2 \times n/2$, porém, ainda assim, somente um número constante de somas. Como antes, o número constante de somas de matrizes será incorporado pela notação Θ quando montarmos a equação de recorrência para caracterizar o tempo de execução.

O método de Strassen não é, de modo algum, óbvio. (Esse talvez seja o maior eufemismo neste livro.) Ele tem quatro etapas:

1. Dividir as matrizes de entrada A e B e a matriz de saída C em submatrizes $n/2 \times n/2$, como na equação (4.9). Essa etapa leva um tempo $\Theta(1)$ por cálculo de índices, exatamente como o procedimento SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Criar 10 matrizes $S_1, S_2, \ldots, S_{10}$, cada uma das quais é $n/2 \times n/2$ e é a soma ou diferença de duas matrizes criadas na etapa 1. Podemos criar todas as 10 matrizes no tempo $\Theta(n_2)$.
3. Usando as submatrizes criadas na etapa 1 e as 10 matrizes criadas na etapa 2, calcular recursivamente sete produtos de matrizes $P_1, P_2, \ldots, P_7$. Cada matriz P_i é $n/2 \times n/2$.
4. Calcular as submatrizes desejadas C_{11}, C_{12}, C_{21}, C_{22} da matriz resultado C somando e subtraindo várias combinações das P_i matrizes. Podemos calcular todas as quatro submatrizes no tempo $\Theta(n_2)$.

Veremos os detalhes das etapas 2-4 em instantes, mas já temos informações suficientes para montar uma recorrência para o tempo de execução do método de Strassen. Vamos considerar que tão logo o tamanho n da matriz atinja 1, efetuamos uma multiplicação escalar simples, exatamente como na linha 4 de SQUARE-MATRIX-MULTIPLY-RECURSIVE. Quando $n > 1$, as etapas 1, 2 e 4 levam um tempo total de $\Theta(n_2)$e a etapa 3 requer que efetuemos sete multiplicações de matrizes $n/2 \times n/2$. Por consequência, obtermos a seguinte recorrência para o tempo de execução $T(n)$ do algoritmo de Strassen:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1; \\ 7T(n/2) + \Theta(n^2) & \text{se } n > 1. \end{cases} \tag{4.18}$$

Trocamos uma multiplicação de matrizes por um número constante de soma de matrizes. Assim que entendermos recorrências e suas soluções, veremos que, na verdade, essa troca resulta em um tempo de execução assintótico mais baixo. Pelo método mestre na Seção 4.5, a solução da recorrência (4.18) é $T(n) = \Theta(n_{\lg} 7)$.

Agora passamos para a descrição dos detalhes. Na etapa 2, criamos as 10 matrizes a seguir:

$$S_1 = B_{12} - B_{22} ,$$
$$S_2 = A_{11} + A_{12} ,$$
$$S_3 = A_{21} + A_{22} ,$$
$$S_4 = B_{21} - B_{11} ,$$
$$S_5 = A_{11} + A_{22} ,$$
$$S_6 = B_{11} + B_{22} ,$$
$$S_7 = A_{12} - A_{22} ,$$
$$S_8 = B_{21} + B_{22} ,$$
$$S_9 = A_{11} - A_{21} ,$$
$$S_{10} = B_{11} + B_{12} .$$

Visto que devemos somar ou subtrair matrizes $n/2 \times n/2$ 10 vezes, essa etapa leva realmente o tempo $\Theta(n_2)$.

Na etapa 3, multiplicamos recursivamente matrizes $n/2 \times n/2$ sete vezes para calcular as seguintes matrizes $n/2 \times n/2$, cada uma das quais é a soma ou a diferença de produtos de submatrizes A e B:

$$P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} ,$$
$$P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} ,$$
$$P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} ,$$
$$P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} ,$$
$$P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} ,$$
$$P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} ,$$
$$P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .$$

Observe que as únicas multiplicações que precisamos executar são as que se encontram na coluna do meio dessas equações. A coluna do lado direito mostra apenas em quê esses produtos são iguais em termos das submatrizes originais criadas na etapa 1.

A etapa 4 soma e subtrai as P_i matrizes criadas na etapa 3 para construir as quatro submatrizes $n/2 \times n/2$ do produto C. Começamos com

$$C_{11} = P_5 + P_4 - P_2 + P_6.$$

Expandindo o lado direito, com a expansão de cada P_i em sua própria linha e alinhando na vertical os termos cancelados, vemos que C_{11} é igual a

$$
\begin{array}{lllllll}
A_{11}\cdot B_{11} & + A_{11}\cdot B_{22} & + A_{22}\cdot B_{11} & + A_{22}\cdot B_{22} & & & \\
 & & - A_{22}\cdot B_{11} & & + A_{22}\cdot B_{21} & & \\
 & - A_{11}\cdot B_{22} & & & & - A_{12}\cdot B_{22} & \\
 & & & - A_{22}\cdot B_{22} & - A_{22}\cdot B_{21} & + A_{12}\cdot B_{22} & + A_{12}\cdot B_{21} \\
\hline
A_{11}\cdot B_{11} & & & & & & + A_{12}\cdot B_{21} ,
\end{array}
$$

que corresponde à equação (4.11).

De modo semelhante, fazemos

$$C_{12} = P_1 + P_2 ,$$

e, portanto, C_{12} é igual a

$$
\begin{array}{llll}
A_{11}\cdot B_{12} & - A_{11}\cdot B_{22} & & \\
 & + A_{11}\cdot B_{22} & + A_{12}\cdot B_{22} & \\
\hline
A_{11}\cdot B_{12} & & + A_{12}\cdot B_{21} ,
\end{array}
$$

correspondendo à equação (4.12).

Fazer

$$C_{21} = P_3 + P_4$$

torna C_{21} igual a

$$
\begin{array}{lll}
A_{21}\cdot B_{11} & + A_{22}\cdot B_{11} & \\
 & - A_{22}\cdot B_{11} & + A_{22}\cdot B_{21} \\
\hline
A_{21}\cdot B_{11} & & + A_{22}\cdot B_{21} ,
\end{array}
$$

correspondente à equação (4.13).

Finalmente, fazemos

$$C_{22} = P_5 + P_1 - P_3 - P_7 ,$$

de modo que C_{22} seja igual a

$$
\begin{array}{lllllll}
A_{11}\cdot B_{11} & + A_{11}\cdot B_{22} & + A_{22}\cdot B_{11} & + A_{22}\cdot B_{22} & & & \\
 & - A_{11}\cdot B_{22} & & & + A_{11}\cdot B_{12} & & \\
 & & - A_{22}\cdot B_{11} & & & - A_{21}\cdot B_{11} & \\
-A_{11}\cdot B_{11} & & & & - A_{11}\cdot B_{12} & + A_{21}\cdot B_{11} & + A_{21}\cdot B_{12} \\
\hline
 & & & A_{22}\cdot B_{22} & & & + A_{21}\cdot B_{12} ,
\end{array}
$$

que corresponde à equação (4.14). No total, somamos ou subtraímos matrizes $n/2 \times n/2$ oito vezes na etapa 4 e, portanto, essa etapa leva de fato o tempo $\Theta(n_2)$.

Assim, vemos que o algoritmo de Strassen, que compreende as etapas 1-4, produz o produto correto de matrizes e que a recorrência (4.18) caracteriza seu tempo de execução. Como veremos na Seção 4.5 que a solução dessa

recorrência é $T(n) = \Theta(n_{\lg}\ 7)$, o método de Strassen é assintoticamente mais rápido do que o procedimento direto SQUARE-MATRIX-MULTIPLY. As notas ao final deste capítulo discutem alguns dos aspectos práticos do algoritmo de Strassen.

Exercícios

Observação: Embora os Exercícios 4.2-3, 4.2-4 e 4.2-5 tratem de variantes do algoritmo de Strassen, você deve ler a Seção 4.5 antes de tentar resolvê-los.

4.2-1 Use o algoritmo de Strassen para calcular o produto de matrizes

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$

Mostre o seu trabalho.

4.2-2 Escreva pseudocódigo para o algoritmo de Strassen.

4.2-3 Como você modificaria o algoritmo de Strassen para multiplicar matrizes $n \times n$ nas quais n não é uma potência exata de 2? Mostre que o algoritmo resultante é executado no tempo $\Theta(n_{\lg}\ 7)$.

4.2-4 Qual é o maior k tal que, se você puder multiplicar matrizes 3×3 usando k multiplicações (sem considerar comutatividade de multiplicação), poderá multiplicar matrizes $n \times n$ no tempo $O(n_{\lg}\ 7)$? Qual seria o tempo de execução desse algoritmo?

4.2-5 V. Pan descobriu um modo de multiplicar matrizes 68×68 usando 132.464 multiplicações, um modo de multiplicar matrizes 70×70 usando 143.640 multiplicações e um modo de multiplicar matrizes 72×72 usando 155.424 multiplicações. Qual é o método que produz o melhor tempo de execução assintótico quando usado em um algoritmo de divisão e conquista para multiplicação de matrizes? Compare-o com o tempo do algoritmo de Strassen.

4.2-6 Com que rapidez é possível multiplicar uma matriz $kn \times n$ por uma matriz $n \times kn$ usando o algoritmo de Strassen como sub-rotina? Responda à mesma pergunta invertendo a ordem das matrizes de entrada.

4.2-7 Mostre como multiplicar os números complexos $a + bi$ e $c + di$ usando apenas três multiplicações de números reais. O algoritmo deve tomar a, b, c e d como entrada e produzir o componente real $ac - bd$ e o componente imaginário $ad + bc$ separadamente.

4.3 MÉTODO DE SUBSTITUIÇÃO PARA RESOLVER RECORRÊNCIAS

Agora, que já vimos como as recorrências caracterizam os tempos dos algoritmos de divisão e conquista, aprenderemos como resolver recorrências. Começamos esta seção com o método de "substituição".

O ***método de substituição*** para resolver recorrências envolve duas etapas:

1. Arriscar um palpite para a forma da solução.
2. Usar indução para determinar as constantes e mostrar que a solução funciona.

Substituímos a função pela solução suposta na primeira etapa quando aplicamos a hipótese indutiva a valores menores; daí o nome "método de substituição". Esse método é poderoso, mas temos de adivinhar a forma da resposta

para aplicá-lo.

Podemos usar o método de substituição para estabelecer limites superiores ou inferiores para uma recorrência. Como exemplo, vamos determinar um limite superior para a recorrência

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \ , \qquad (4.19)$$

que é semelhante às recorrências (4.3) e (4.4). Arriscamos o palpite de que a solução é $T(n) = O(n \lg n)$. O método de substituição requer que provemos que $T(n) \le cn \lg n$ para uma escolha apropriada da constante $c > 0$. Começamos considerando que esse limite se mantém válido para todo $m < n$ positivo, em particular para $n/2$, o que produz $T(n/2) \le c\, n/2 \lg(n/2)$. Substituindo na recorrência obtemos

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n \ , \end{aligned}$$

onde a última etapa é válida desde que $c \ge 1$.

Agora, a indução exige que mostremos que nossa solução se mantém válida para as condições de contorno. Normalmente, fazemos isso mostrando que as condições de contorno são adequadas como casos-base para a prova indutiva. No caso da recorrência (4.19), devemos mostrar que podemos escolher a constante c suficientemente grande de modo que o limite $T(n) \le cn \lg n$ também funcione para as condições de contorno. Às vezes, essa exigência pode gerar problemas. Vamos supor, como argumento, que $T(1) = 1$ seja a única condição de contorno da recorrência. Então, para $n = 1$, o limite $T(n) \le cn \lg n$ produz $T(1) \le c\ 1 \lg 1 = 0$, o que está em desacordo com $T(1) = 1$. Consequentemente, o caso-base de nossa prova indutiva deixa de ser válido.

Podemos superar esse obstáculo para provar uma hipótese indutiva para uma condição de contorno específica com apenas um pouco mais de esforço. Por exemplo, na recorrência (4.19), tiramos proveito de notação assintótica que só exige que provemos que $T(n) \le cn \lg n$ para $n \ge n_0$, onde n_0 é uma constante *de nossa escolha.* Mantemos a importuna condição de contorno $T(1) = 1$, mas não a consideramos na prova indutiva. Fazemos isso primeiro observando que, para $n > 3$, a recorrência não depende diretamente de $T(1)$. Desse modo, podemos substituir $T(1)$ por $T(2)$ e $T(3)$ como os casos-base na prova indutiva fazendo $n_0 = 2$. Observe que fazemos uma distinção entre o caso-base da recorrência ($n = 1$) e os casos-base da prova indutiva ($n = 2$ e $n = 3$). Com $T(1) = 1$, derivamos da recorrência que $T(2) = 4$ e $T(3) = 5$. Agora podemos concluir a prova indutiva de $T(n) \le cn \lg n$ para alguma constante $c \ge 1$ escolhendo c suficientemente grande de modo que $T(2) \le c\ 2 \lg 2$ e $T(3) \le c\ 3 \lg 3$. Como observamos, qualquer valor de $c \ge 2$ é suficiente para que os casos-base de $n = 2$ e $n = 3$ sejam válidos. Para a maioria das recorrências que examinaremos, estender as condições de contorno para fazer a hipótese indutiva funcionar para n pequeno é um procedimento direto, e nem sempre elaboraremos explicitamente os detalhes.

Como dar um bom palpite

Infelizmente, não há nenhum modo geral para adivinhar as soluções corretas para recorrências. Arriscar um palpite para uma solução exige experiência e, ocasionalmente, criatividade. Entretanto, por sorte, você pode usar a heurística para ajudá-lo a se tornar um bom adivinhador. Além disso, poderá também usar árvores de recursão, que veremos na Seção 4.4, para gerar bons palpites.

Se uma recorrência for semelhante a alguma que você já tenha visto antes, será razoável supor uma solução semelhante. Como exemplo, considere a recorrência

$T(n) = 2T(n/2 + 17) + n$,

que parece difícil devido ao "17" acrescentado ao argumento de T no lado direito. Porém, intuitivamente, esse termo adicional não pode afetar substancialmente a solução para a recorrência. Quando n é grande, a diferença entre $n/2$ e $n/2 + 17$ não é tão grande: ambos cortam n quase uniformemente pela metade. Em consequência disso, arriscamos $T(n) = O(n \lg n)$, o que você pode verificar que é correto usando o método de substituição (veja Exercício 4.3.6).

Um outro modo de dar um bom palpite é comprovar limites superiores e inferiores frouxos para a recorrência e, então, reduzir a faixa de incerteza. Por exemplo, poderíamos começar com um limite inferior de $T(n) = (n)$ para a recorrência (4.19), já que temos o termo n na recorrência e podemos comprovar um limite superior inicial de $T(n) = O(n_2)$. Então, podemos diminuir gradualmente o limite superior e elevar o limite inferior, até convergirmos na solução correta, assintoticamente justa, $T(n) = \Theta(n \lg n)$.

Sutilezas

Às vezes, você pode dar um palpite correto para um limite assintótico para a solução de uma recorrência mas, por alguma razão, a matemática não consegue funcionar na indução. Em geral, observamos que a hipótese indutiva não é suficientemente forte para comprovar o limite apontado. Se você revisar seu palpite subtraindo um termo de ordem mais baixa quando chegar a um impasse como esse, a matemática frequentemente funcionará.

Considere a recorrência

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 .$$

Adivinhamos que a solução é $T(n) = O(n)$, e tentamos mostrar que $T(n) \leq cn$ para uma escolha adequada da constante c. Substituindo nosso palpite na recorrência, obtemos

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 , \end{aligned}$$

o que não implica $T(n) \leq cn$ para qualquer escolha de c. Poderíamos ficar tentados a experimentar um palpite maior, digamos $T(n) = O(n_2)$. Embora possamos fazer esse palpite maior funcionar, nosso palpite original $T(n) = O(n)$ é correto. Porém, para mostrar que ele é correto, temos de fazer uma hipótese indutiva mais forte.

Intuitivamente, nosso palpite quase correto é: a única diferença é a constante 1, um termo de ordem mais baixa. Apesar disso, a indução não funciona, a menos que provemos a forma exata da hipótese indutiva. Superamos nossa dificuldade *subtraindo* um termo de ordem mais baixa de nosso palpite anterior. Nosso novo palpite é $T(n) \leq cn - d$, onde $d \geq 0$ é uma constante. Agora temos

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \\ &\leq cn - d, \end{aligned}$$

desde que $d \geq 1$. Como antes, devemos escolher a constante c suficientemente grande para lidar com as condições de contorno.

Você pode achar que a ideia de subtrair um termo de ordem mais baixa é anti-intuitiva. Afinal, se a matemática não funciona, o certo não seria aumentar nosso palpite? Não necessariamente! Quando queremos provar um limite superior por indução, na verdade pode ser mais difícil provar que um limite inferior mais fraco é válido porque, para provar um limite mais fraco, temos de usar na prova, indutivamente, o mesmo limite mais fraco. Nesse nosso exemplo, quando a recorrência tem mais de um termo recursivo, temos de subtrair o termo de ordem mais baixa do limite proposto uma vez por termo recursivo. No exemplo anterior, subtraímos a constante d duas vezes, uma vez para o termo $T(n/2)$ e uma vez para o termo $T(n/2)$. Terminamos com a desigualdade $T(n) \leq cn - 2d + 1$, e foi fácil encontrar valores de d que tornassem $cn - 2d + 1$ menor ou igual a $cn - d$.

Como evitar armadilhas

É fácil errar na utilização da notação assintótica. Por exemplo, na recorrência (4.19), podemos "provar" falsamente que $T(n) = O(n)$, supondo $T(n) \leq cn$ e, então, argumentando que

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n) \, , \qquad \Leftarrow \textit{errado!!} \end{aligned}$$

visto que c é uma constante. O erro é que não provamos a *forma exata* da hipótese indutiva, isto é, que $T(n) \leq cn$. Portanto, provaremos explicitamente que $T(n) \leq cn$ quando quisermos mostrar que $T(n) = O(n)$.

Como trocar variáveis

Às vezes, um pouco de manipulação algébrica pode tornar uma recorrência desconhecida semelhante a alguma que você já viu antes. Como exemplo, considere a recorrência

$$T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \lg n,$$

que parece difícil. Entretanto, podemos simplificar essa recorrência com uma troca de variáveis. Por conveniência, não nos preocuparemos com arredondar valores, como $\sqrt{n}$, para que fiquem inteiros. Renomear $m = \lg n$ produz

$$T(2^m) = 2T\,(2^{m/2}) + m \, .$$

Agora podemos renomear $S(m) = T(2^m)$ para produzir a nova recorrência

$$S(m) = 2S(m/2) + m \, ,$$

que é muito semelhante à recorrência (4.19). De fato, essa nova recorrência tem a mesma solução: $S(m) = O(m \lg m)$. Voltando a trocar $S(m)$ por $T(n)$, obtemos $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

Exercícios

4.3-1 Mostre que a solução de $T(n) = T(n - 1) + n$ é $O(n_2)$.

4.3-2 Mostre que a solução de $T(n) = T(\ n/2) + 1$ é $O(\lg n)$.

4.3-3 Vimos que a solução de $T(n) = 2T(n/2) + n$ é $O(n \lg n)$. Mostre que a solução dessa recorrência é também $(n \lg n)$. Conclua que a solução é $\Theta(n \lg n)$.

4.3-4 Mostre que, formulando uma hipótese indutiva diferente, podemos superar a dificuldade com a condição de contorno $T(1) = 1$ para a recorrência (4.19) sem ajustar as condições de contorno para a prova indutiva.

4.3-5 Mostre que $\Theta(n \lg n)$ é a solução para a recorrência "exata" (4.3) para a ordenação por intercalação.

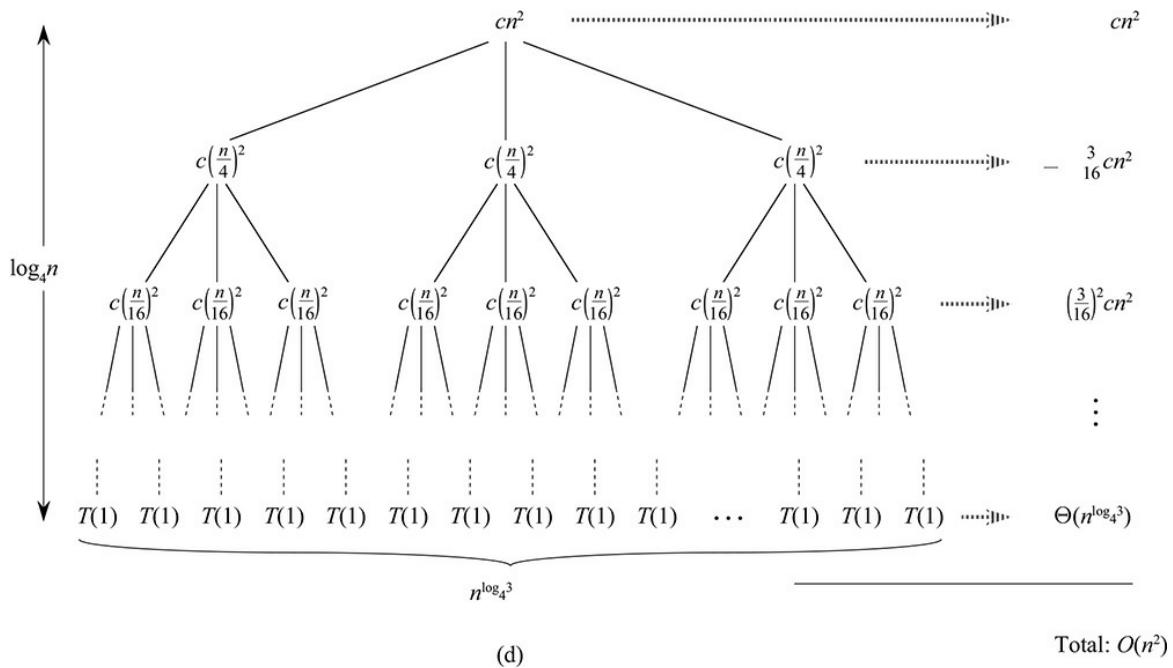
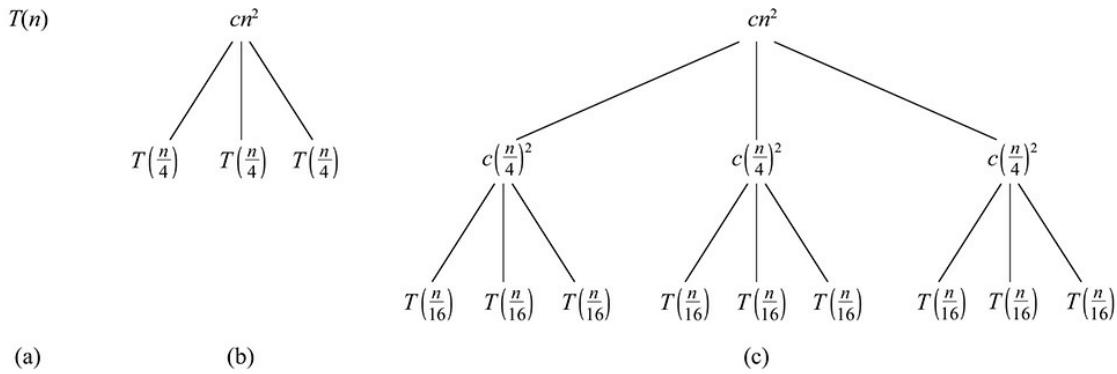
4.3-6 Mostre que a solução para $T(n) = 2T(n/2 + 17) + n$ é $O(n \lg n)$.

4.3-7 Usando o método mestre da Seção 4.5, você pode mostrar que a solução para a recorrência $T(n) = 4T(n/3) + n$ é $T(n) = \Theta(n_{\log\,3}4)$. Mostre que uma prova de substituição que considere $T(n) \leq cn_{\log}\,4$ falha. Então, mostre como subtrair um termo de ordem mais baixa para fazer com que uma prova de substituição funcione.

4.3-8 Usando o método mestre da Seção 4.5, você pode mostrar que a solução para a recorrência $T(n) = 4T(n/2) + n$ é $T(n) = \Theta(n_2)$. Mostre que uma prova de substituição que considere $T(n) \leq cn_2$ falha. Então, mostre como subtrair um termo de ordem mais baixa para fazer com que uma prova de substituição funcione.

4.3-9 Resolva a recorrência $T(n) = 3T(\sqrt{n}\ ) + \log n$ fazendo uma troca de variáveis. Sua solução deve ser assintoticamente justa. Não se preocupe com saber se os valores são inteiros.

4.4 MÉTODO DA ÁRVORE DE RECURSÃO PARA RESOLVER RECORRÊNCIAS

Embora você possa usar o método de substituição para obter uma prova sucinta de que uma solução para uma recorrência é correta, às vezes, é difícil apresentar um bom palpite. Traçar uma árvore de recursão, como fizemos em nossa análise da recorrência da ordenação por intercalação na Seção 2.3.2, é um modo direto para dar um bom palpite. Em uma ***árvore de recursão***, cada nó representa o custo de um único subproblema em algum lugar no conjunto de invocações de função recursiva. Somamos os custos em cada nível da árvore para obter um conjunto de custos por nível e depois somamos todos os custos por nível para determinar o custo total de todos os níveis da recursão.

Uma árvore de recursão é mais bem usada para gerar um bom palpite, que é então verificado pelo método de substituição. Ao usar uma árvore de recursão para gerar um bom palpite, muitas vezes você pode tolerar um pouco de "desleixo", já que verificará seu palpite mais tarde. Porém, se você for muito cuidadoso ao desenhar uma árvore de recursão e somar os custos, poderá usar uma árvore de recursão como prova direta de uma solução para uma recorrência. Nesta seção, usaremos árvores de recursão para gerar bons palpites e, na Seção 4.6, utilizaremos árvores de recursão diretamente para provar o teorema que forma a base do método mestre.

Por exemplo, vejamos como uma árvore de recursão daria um bom palpite para a recorrência $T(n) = 3T(n/4) + \Theta(n_2)$. Começamos focalizando a determinação de um limite superior para a solução. Como sabemos que pisos e tetos normalmente não têm importância na solução de recorrências (esse é um exemplo de desleixo que podemos tolerar), criamos uma árvore de recursão para a recorrência $T(n) = 3T(n/4) + cn_2$, tendo explicitado o coeficiente constante implícito $c > 0$.

A Figura 4.5 mostra como derivamos a árvore de recursão para $T(n) = 3T(n/4) + cn_2$. Por conveniência, supomos que n é uma potência exata de 4 (outro exemplo de desleixo tolerável) de modo que os tamanhos de todos os subproblemas são inteiros. A parte (a) da figura mostra $T(n)$ que, na parte (b), expandimos para uma árvore equivalente que representa a recorrência. O termo cn_2 na raiz representa o custo no nível superior da recursão, e as três subárvores da raiz representam os custos incorridos pelos subproblemas de tamanho $n/4$. A parte (c) mostra a continuação desse processo em uma etapa posterior representada pela expansão de cada nó com custo $T(n/4)$ da parte (b). O custo para cada um dos três filhos da raiz é $c(n/4)^2$. Continuamos a expandir cada nó na árvore, desmembrando-o em suas partes constituintes conforme determinado pela recorrência.

Visto que os tamanhos dos subproblemas diminuem por um fator de 4 toda vez que descemos um nível, a certa altura devemos alcançar uma condição de contorno. A que distância da raiz nós a encontramos? O tamanho do subproblema para um nó na profundidade i é $n/4^i$. Desse modo, o tamanho do subproblema chega a $n = 1$ quando $n/4^i = 1$ ou, o que é equivalente, quando $i = \log^4 n$. Assim, a árvore tem $\log^4 n + 1$ níveis (nas profundidades 0, 1, 2, . .., $\log^4 n$).

Em seguida, determinamos o custo em cada nível da árvore. Cada nível tem três vezes mais nós que o nível acima dele, portanto o número de nós na profundidade i é 3^i. Como os tamanhos dos subproblemas se reduzem por um fator de 4 para cada nível que descemos a partir da raiz, cada nó na profundidade i, para $i = 0, 1, 2, . .., \log^4 n - 1$, tem o custo de $c(n/4^i)^2$. Multiplicando, vemos que o custo total para todos os nós na profundidade i, para $i = 0, 1, 2, . .., \log^4 n - 1$, é $3^i c(n/4^i)^2 = (3/16)^i cn_2$. O nível inferior, na profundidade $\log_4 n$, tem $3^{\log 4_{\mathrm{n}}} = n_{\log}{}^{43}$ nós, e cada um deles contribui com o custo $T(1)$, para um custo total de $n_{\log}{}^{43}T(1)$, que é $\Theta(n_{\log}{}^{43})$, já que supomos que $T(1)$ é uma constante.

Figura 4.5 Construção de uma árvore de recursão para a recorrência $T(n) = 3T(n/4) + cn_2$. A parte **(a)** mostra $T(n)$, que expande-se progressivamente em **(b)-(d)** para formar a árvore de recursão. A árvore completamente expandida na parte (d) tem altura $\log_4 n$ (ela tem $\log_4 n + 1$ níveis).

Agora somamos os custos em todos os níveis para determinar o custo da árvore inteira:

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \qquad \text{(pela equação (A.5)).}
\end{aligned}
$$

Esta última fórmula parece um pouco confusa até percebermos que, mais uma vez, é possível tirar proveito de uma certo desleixo e usar uma série geométrica decrescente infinita como um limite superior. Retrocedendo uma etapa e

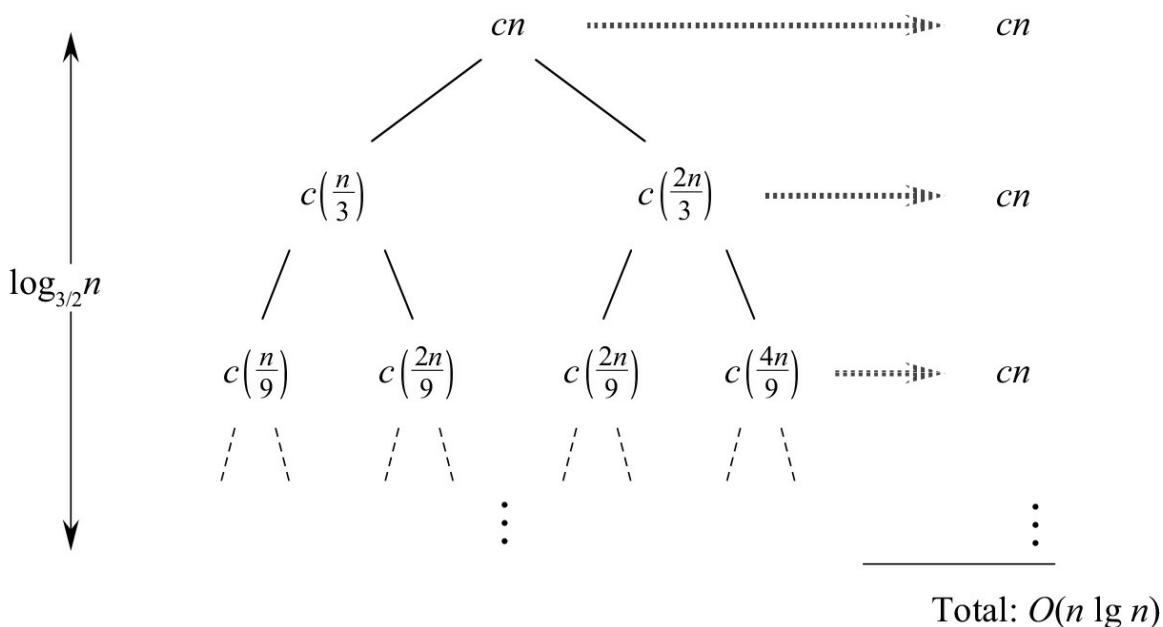
aplicando a equação (A.6), temos

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2+\Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2+\Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)}cn^2+\Theta(n^{\log_4 3}) \\
&= \frac{16}{13}cn^2+\Theta(n^{\log_4 3}) \\
&= O(n^2).
\end{aligned}
$$

Assim, derivamos um palpite de $T(n)=O(n_2)$ para nossa recorrência original $T(n)=3T(\mathrm{n}/4)+\Theta(n_2)$. Nesse exemplo, os coeficientes de cn_2 formam uma série geométrica decrescente e, pela equação (A.6), a soma desses coeficientes é limitada na parte superior pela constante 16/13. Visto que a contribuição da raiz para o custo total é cn_2, a raiz contribui com uma fração constante do custo total. Em outras palavras, o custo da raiz domina o custo total da árvore.

De fato, se $O(n_2)$ é realmente um limite superior para a recorrência (como verificaremos em breve), ele deve ser um limite justo. Por quê? A primeira chamada recursiva contribui com o custo $\Theta(n_2)$, então (n_2) deve ser um limite inferior para a recorrência.

Agora podemos usar o método de substituição para verificar que nosso palpite era correto, isto é, $T(n) = O(n_2)$ é um limite superior para a recorrência $T(n) = 3T(n/4) + \Theta(n_2)$. Queremos mostrar que $T(n) \leq dn_2$ para alguma constante $d > 0$. Usando a mesma constante $c > 0$ de antes, temos

$$
\begin{aligned}
T(n) &\leq 3T(\lfloor n/4 \rfloor)+cn^2 \\
&\leq 3d\lfloor n/4 \rfloor^2+cn^2 \\
&\leq 3d(n/4)^2+cn^2 \\
&= \frac{3}{16}dn^2+cn^2 \\
&\leq dn^2,
\end{aligned}
$$

onde a última etapa é válida desde que $d \geq (16/13)c$.

Em outro exemplo mais complicado, a Figura 4.6 mostra a árvore de recursão para

$$T(n) = T(n/3) + T(2n/3) + O(n) .$$

(Novamente, omitimos as funções piso e teto por simplicidade.) Como antes, c representa o fator constante no termo $O(n)$. Quando somamos os valores em todos os níveis da árvore de recursão, obtemos um valor de cn para cada nível. O caminho simples mais longo da raiz até uma folha é $n \to (2/3)n \to (2/3)^2 n \to \dots \to 1$. Visto que $(2/3)^k n = 1$ quando $k = \log_{3/2} n$, a altura da árvore é $\log_{3/2} n$

Intuitivamente, esperamos que a solução para a recorrência seja, no máximo, o número de níveis vezes o custo de cada nível ou $O(cn \log_{3/2} n) = O(n \lg n)$. Entretanto, a Figura 4.6 mostra apenas os níveis superiores da árvore de recursão, e nem todo nível da árvore contribui com um custo cn. Considere o custo das folhas. Se essa árvore de recursão fosse uma árvore binária completa de altura $\log_{3/2} n$, haveria $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ folhas. Como o custo de cada folha é uma constante, o custo total de todas as folhas será $\Theta(n^{\log_{3/2} 2})$ que, visto que $\log_{3/2} 2$ é uma constante estritamente maior do que 1, é $(n \lg n)$. Contudo, essa árvore de recursão não é uma árvore binária completa e, por isso, ela tem menos de $n^{\log_{3/2} 2}$ folhas. Além do mais, à medida que descemos em relação à raiz, mais e mais nós internos estão ausentes. Consequentemente, níveis mais próximos da parte inferior da árvore de recursão contribuem com menos de cn para o custo total. Poderíamos desenvolver uma contabilidade precisa de todos os custos, mas lembre-se de que estamos apenas tentando propor um palpite para usar no método de substituição. Vamos tolerar o desleixo e tentar mostrar que um palpite $O(n \lg n)$ para o limite superior é correto.

Figura 4.6 Uma árvore de recursão para a recorrência $T(n) = T(n/3) + T(2n/3) + cn$.

De fato, podemos usar o método de substituição para verificar que $O(n \lg n)$ é um limite superior para a solução da recorrência. Mostramos que $T(n) \leq dn \lg n$, onde d é uma constante positiva adequada. Temos

$$
\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\qquad + (d(2n/3)\lg n - d(2n/3)\lg(3/2) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\le dn\lg n,
\end{aligned}
$$

desde que $d \geq c/(\lg 3 - (2/3))$. Assim, não tivemos de executar uma contabilidade de custos mais precisa na árvore de recursão.

Exercícios

4.4-1 Use uma árvore de recursão para determinar um bom limite superior assintótico para a recorrência $T(n) = 3T(n/2) + n$. Use o método de substituição para verificar sua resposta.

4.4-2 Use uma árvore de recursão para determinar um bom limite superior assintótico para a recorrência $T(n) = T(n/2) + n_2$. Use o método de substituição para verificar sua resposta.

4.4-3 Use uma árvore de recursão para determinar um bom limite superior assintótico para a recorrência $T(n) = 4T(n/2 + 2) + n$. Use o método de substituição para verificar sua resposta.

4.4-4 Use uma árvore de recursão para determinar um bom limite superior assintótico para a recorrência $T(n) = 2T(n - 1) + 1$. Use o método de substituição para verificar sua resposta.

4.4-5 Use uma árvore de recursão para determinar um bom limite superior assintótico para a recorrência $T(n) = T(n - 1) + T(n/2) + n$. Use o método de substituição para verificar sua resposta.

4.4-6 Demonstre que a solução para a recorrência $T(n) = T(n/3) + T(2n/3) + cn$, onde c é uma constante, é $(n \lg n)$, apelando para uma árvore de recursão.

4.4-7 Trace a árvore de recursão para $T(n) = 4T(n/2) + cn$, onde c é uma constante, e forneça um limite assintótico restrito para a sua solução. Verifique o limite pelo método de substituição.

4.4-8 Use uma árvore de recursão para dar uma solução assintoticamente justa para a recorrência $T(n) = T(n - a) + T(a) + cn$, onde $a \geq 1$ e $c > 0$ são constantes.

4.4-9 Use uma árvore de recursão para dar uma solução assintoticamente justa para a recorrência $T(n) = T(\infty n) + T((1 - \alpha)n) + cn$, onde a é uma constante no intervalo $0 < \alpha < 1$ e $c > 0$ também é uma constante.

4.5 MÉTODO MESTRE PARA RESOLVER RECORRÊNCIAS

O método mestre fornece uma "receita" para resolver recorrências da forma

$$T(n) = aT(n/b) + f(n)\,, \tag{4.20}$$

onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva. Para utilizar o método mestre você terá de memorizar três casos, mas poderá resolver muitas recorrências com grande facilidade, muitas vezes sem lápis e papel.

A recorrência (4.20) descreve o tempo de execução de um algoritmo que divide um problema de tamanho n em a subproblemas, cada um de tamanho n/b, onde a e b são constantes positivas. Os a subproblemas são resolvidos recursivamente, cada um no tempo $T(n/b)$. A função $f(n)$ abrange o custo de dividir o problema e combinar os resultados dos subproblemas. Por exemplo, a recorrência que surge do algoritmo de Strassen tem $a = 7$, $b = 2$, e $f(n) = \Theta(n_2)$.

Por questão de correção técnica, na realidade a recorrência não está bem definida porque n/b poderia não ser um inteiro. Porém, substituir cada um dos a termos $T(n/b)$ por $T(n/b)$ ou $T(n/b)$ não afetará o comportamento assintótico da recorrência. (Provaremos essa afirmação na próxima seção.) Portanto, normalmente consideramos conveniente omitir as funções piso e teto quando escrevemos recorrências de divisão e conquista dessa forma.

O teorema mestre

O método mestre depende do teorema a seguir.

Teorema 4.1 (Teorema mestre)

Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida no domínio dos números inteiros não negativos pela recorrência

$$T(n) = aT(n/b) + f(n),$$

onde interpretamos que n/b significa n/b oun/b. Então, $T(n)$ tem os seguintes limites assintóticos:

1. Se $f(n) = O(n^{\log_b a -})$ para alguma constante > 0, então $T(n) = \Theta(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Se $f(n) = (n^{\log_b a +})$ para alguma constante > 0, e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todos os n suficientemente grandes, então $T(n) = \Theta(f(n))$.

Antes de aplicar o teorema mestre a alguns exemplos, vamos dedicar algum tempo tentando entender o que ele significa. Em cada um dos três casos, comparamos a função $f(n)$ com a função $n^{\log_b a}$. Intuitivamente, a maior das duas funções determina a solução para a recorrência. Se, como no caso 1, a função $n^{\log_b a}$ for a maior, então a solução é $T(n) = \Theta(n^{\log_b a})$. Se, como no caso 3, a função $f(n)$ for a maior, então a solução é $T(n) = \Theta(f(n))$. Se, como no caso 2, as duas funções tiverem o mesmo tamanho, multiplicamos por um fator logarítmico e a solução é $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Além dessa intuição, você precisa estar ciente de alguns detalhes técnicos. No primeiro caso, $f(n)$ não só tem de ser menor que $n^{\log_b a}$, mas deve ser *polinomialmente* menor. Isto é, $f(n)$ deve ser assintoticamente menor que $n^{\log_b a}$ por um fator n para alguma constante > 0. No terceiro caso, $f(n)$ não apenas deve ser maior que $n^{\log_b a}$, ela tem de ser polinomialmente maior e, além disso, satisfazer à condição de "regularidade" expressa por $af(n/b) \leq cf(n)$. Essa condição é satisfeita pela maioria das funções polinomialmente limitadas que encontraremos.

Observe que os três casos não abrangem todas as possibilidades para $f(n)$. Existe uma lacuna entre os casos 1 e 2 quando $f(n)$ é menor que $n^{\log_b a}$, mas não polinomialmente menor. De modo semelhante, há uma lacuna entre os casos 2 e 3 quando $f(n)$ é maior que $n^{\log_b a}$, mas não polinomialmente maior. Se a função $f(n)$ cair em uma dessas lacunas ou se a condição de regularidade no caso 3 deixar de ser válida, o método mestre não poderá ser usado para resolver a recorrência.

Como usar o método mestre

Para usar o método mestre, simplesmente determinamos qual caso (se houver algum) do teorema mestre se aplica e anotamos a resposta.

Como primeiro exemplo, considere

$T(n) = 9T(n/3) + n.$

Para essa recorrência, temos $a = 9$, $b = 3$, $f(n) = n$ e, portanto, temos que $n_{\log_b a} = n_{\log_3 9} = \Theta(n_2)$. Visto que $f(n)$, onde = 1, podemos aplicar o caso 1 do teorema mestre e concluir que a solução é $T(n) = \Theta(n_2)$.

Agora, considere

$T(n) = T(2n/3) + 1,$

na qual $a = 1$, $b = 3/2$, $f(n) = 1$ e $n_{\log_b a} = n_{\log_{3/2} 1} = n_0 = 1$. Aplica-se o caso 2, já que $f(n) = \Theta(n_{\log_b a}) = \Theta(1)$ e, assim, a solução para a recorrência é $T(n) = \Theta(\lg n)$.

Para a recorrência

$T(n) = 3T(n/4) + n \lg n$,

temos $a = 3$, $b = 4$, $f(n) = n \lg n$ e $n_{\log_b a} = n_{\log_4 3} = O(n_{0},^{793})$. Visto que $f(n) = (n_{\log_4 3+})$, onde $\approx 0{,}2$, aplicamos o caso 3 se pudermos mostrar que a condição de regularidade é válida para $f(n)$. Para n suficientemente grande, temos que $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$ para $c = 3/4$. Consequentemente, pelo caso 3, a solução para a recorrência é $T(n) = \Theta(n \lg n)$.

O método mestre não se aplica à recorrência

$T(n) = 2T(n/2) + n \lg n,$

ainda que aparentemente ela tenha a forma apropriada: $a = 2$, $b = 2$, $f(n) = n \lg n$ e $n_{\log_b a} = n$. Você poderá se enganar e achar que o caso 3 deve se aplicar, já que $f(n) = n \lg n$ é assintoticamente maior que $n_{\log_b a} = n$. O problema é que ela não é *polinomialmente* maior. A razão $f(n)/n_{\log_b a} = (n \lg n)/n = \lg n$ é assintoticamente menor que n para qualquer constante positiva . Consequentemente, a recorrência cai na lacuna entre o caso 2 e o caso 3. (Veja uma solução no Exercício 4.6-2.)

Vamos usar o método mestre para resolver as recorrências que vimos nas Seções 4.1 e 4.2. A recorrência (4.7),

$T(n) = 2T(n/2) + \Theta(n)$,

caracteriza os tempos de execução do algoritmo de divisão e conquista para o problema do subarranjo máximo e também para a ordenação por intercalação. (Como de praxe, omitiremos a declaração do caso-base na recorrência.) Aqui, temos $a = 2$, $b = 2$, $f(n) = \Theta(n)$ e, assim, temos que $n_{\log_b a} = n_{\log_2 2} = n$. O caso 2 se aplica, visto que $f(n) = \Theta(n)$ e, portanto, temos a solução $T(n) = \Theta(n \lg n)$.

A recorrência (4.17),

$T(n) = 8T(n/2) = \Theta(n_2),$

descreve o tempo de execução do primeiro algoritmo de divisão e conquista que vimos para multiplicação de matrizes. Agora, temos $a = 8$, $b = 2$ e $f(n) = \Theta(n_2)$ e, assim, $n_{\log_b a} = n_{\log_2 8} = n_3$. Visto que n_3 é polinomialmente maior que $f(n)$ (isto é, $f(n) = O(n_3-)$ para = 1), o caso 1 se aplica e $T(n) = \Theta(n_3)$.

Finalmente, considere a recorrência (4.18),

$T(n) = 7T(n/2) + \Theta(n_2)$,

que descreve o tempo de execução do algoritmo de Strassen. Aqui, temos $a = 7$, $b = 2$, $f(n) = \Theta\ (n_2)$ e, assim, $n_{\log_b a} = n_{\log_2 7}$. Reescrevendo $\log_2 7$ como $\lg 7$ e lembrando que $2{,}80 < \lg 7 < 2{,}81$, vemos que $f(n) = O(n_{\lg\ 7}-)$ para = 0,8. Novamente, o caso 1 se aplica e temos a solução $T(n) = \Theta(n_{\lg\ 7})$.

Exercícios

4.5-1 Use o método mestre para fornecer limites assintóticos restritos para as recorrências a seguir.

- ***a.*** $T(n) = 2T(n/4) + 1$.
- ***b.*** $T(n) = 2T(n/4) + \sqrt{n}$.
- ***c.*** $T(n) = 2T(n/4) + n$.
- ***d.*** $T(n) = 2T(n/4) + n_2$.

4.5-2 O professor César quer desenvolver um algoritmo para multiplicação de matrizes que seja assintoticamente mais rápido do que o algoritmo de Strassen. Seu algoritmo usará o método de divisão e conquista, repartindo cada matriz em pedaços de tamanho $n/4 \times n/4$, e, juntas, as etapas de dividir e combinar levarão o tempo $\Theta(n_2)$. Ele precisa determinar quantos subproblemas tal algoritmo tem de criar para superar o algoritmo de Strassen. Se o algoritmo criar α subproblemas, a recorrência para o tempo de execução $T(n)$ se tornará $T(n) = aT(n/4) + \Theta(n_2)$. Qual é o maior valor inteiro de a para o qual o algoritmo do professor César seria assintoticamente mais rápido que o algoritmo de Strassen?

4.5-3 Use o método mestre para mostrar que a solução para a recorrência de busca binária $T(n) = T(n/2) + \Theta(1)$ é $T(n) = \Theta(\lg n)$. (Veja no Exercício 2.3-5 uma descrição da busca binária.)

4.5-4 O método mestre pode ser aplicado à recorrência $T(n) = 4T(n/2) + n_2\ lg\ n$? Justifique sua resposta. Dê um limite superior assintótico para essa recorrência.

4.5-5 ★

Considere a condição de regularidade $af(n/b) \leq cf(n)$ para alguma constante $c < 1$, que faz parte do caso 3 do teorema mestre. Dê um exemplo de constantes $a \geq 1$ e $b > 1$ e uma função f(n) que satisfaçam todas as condições no caso 3 do teorema mestre, exceto a condição de regularidade.

4.6 ★ PROVA DO TEOREMA MESTRE

Esta seção contém uma prova do teorema mestre (Teorema 4.1). A prova não precisa ser entendida para se aplicar o teorema mestre.

A prova tem duas partes. A primeira parte analisa a recorrência mestre (4.20), adotando a premissa simplificadora de que $T(n)$ é definida apenas para potências exatas de $b > 1$, isto é, para $n = 1, b, b_2, \ldots$. Essa parte nos dá toda a intuição necessária para entender por que o teorema mestre é verdadeiro. A segunda parte mostra como estender a análise a todos os inteiros positivos n; trata com técnica matemática o problema do tratamento de pisos e tetos.

Nesta seção, algumas vezes abusaremos um pouco de nossa notação assintótica usando-a para descrever o comportamento de funções que são definidas somente para potências exatas de b. Lembre-se de que as definições de notações assintóticas exigem que os limites sejam provados para todos os números suficientemente grandes, não apenas para aqueles que são potências de b. Visto que poderíamos produzir novas notações assintóticas que se aplicassem somente ao conjunto $\{b_i : i = 0, 1, \ldots\}$ em vez de aos números não negativos, esse abuso é de menor importância.

Apesar disso, sempre deveremos estar atentos quando usarmos a notação assintótica em um domínio limitado, para não chegarmos a conclusões inadequadas. Por exemplo, provar que $T(n) = O(n)$ quando n é uma potência exata de 2 não garante que $T(n) = O(n)$. A função $T(n)$ poderia ser definida como

$$T(n) = \begin{cases} n & \text{se } n = 1, 2, 4, 8, \ldots, \\ n^2 & \text{senão.} \end{cases}$$

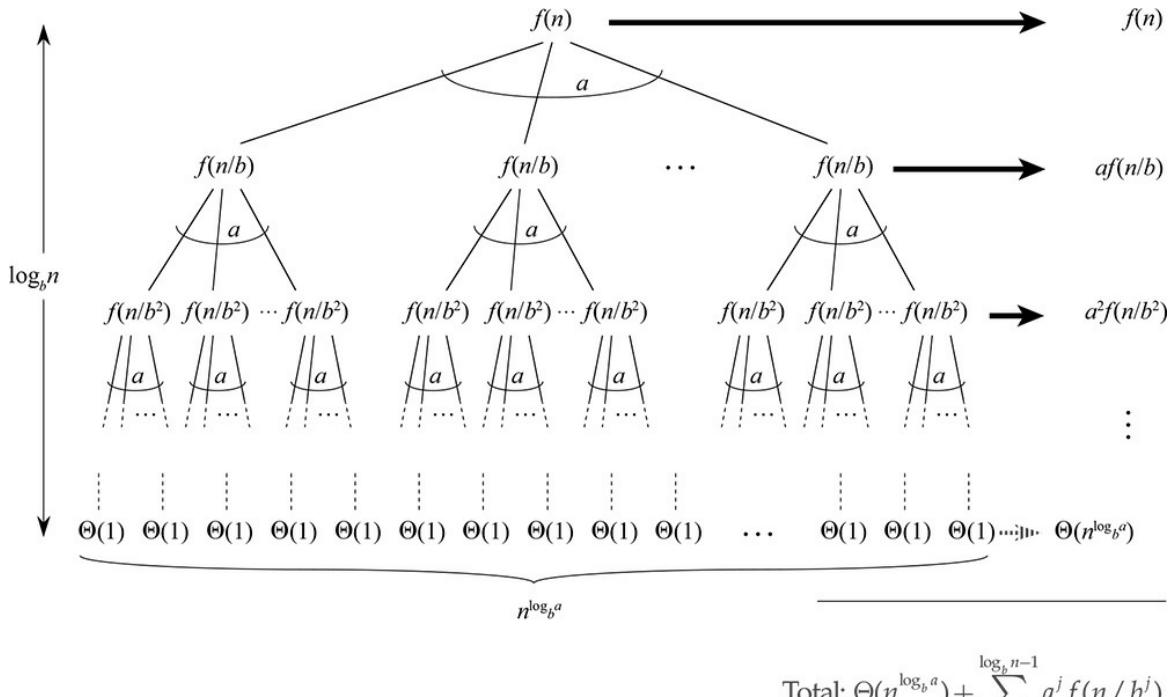
e, nesse caso, o melhor limite superior que se aplica a todos os valores de n é $T(n) = O(n_2)$. Devido a esse tipo de consequência drástica, nunca empregaremos a notação assintótica a um domínio restrito sem deixar absolutamente claro pelo contexto que estamos fazendo isso.

4.6.1 A PROVA PARA POTÊNCIAS EXATAS

A primeira parte da prova do teorema mestre analisa a recorrência (4.20)

$$T(n) = aT(n/b) + f(n) ,$$

para o método mestre, adotando a premissa de que n é uma potência exata de $b > 1$, onde b não precisa ser um inteiro. Dividimos a análise em três lemas. O primeiro reduz o problema de resolver a recorrência mestre ao problema de avaliar uma expressão que contém um somatório. O segundo determina limites para esse somatório. O terceiro lema reúne os dois primeiros para provar uma versão do teorema mestre para o caso em que n é uma potência exata de b.

Lema 4.2

Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida para potências exatas de b. Defina $T(n)$ para potências exatas de b pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1, \\ aT(n/b) + f(n) & \text{se } n = b^i, \end{cases}$$

onde i é um inteiro positivo. Então,

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j). \tag{4.21}$$

Figura 4.7 A árvore de recursão gerada por $T(n) = aT(n/b) + f(n)$. Essa é uma árvore a-ária completa com $n^{\log_b a}$ folhas e altura $\log_b n$. O custo dos nós em cada profundidade é mostrado à direita, e sua soma é dada na equação (4.21).

Prova Usamos a árvore de recursão da Figura 4.7. A raiz da árvore tem custo $f(n)$, e ela tem a filhas, cada uma com custo $f(n/b)$. (É conveniente imaginar a como um inteiro, especialmente se visualizamos a árvore de recursão, mas a matemática não o exige.) Cada uma dessas filhas tem a filhas, o que resulta em a_2 nós na profundidade 2, e cada uma das a filhas tem custo $f(n/\ b_2)$. Em geral, há a_j nós à profundidade j, e cada um tem o custo $f(n/b_j)$. O custo de cada folha é $T(1) = \Theta(1)$, e cada folha está a uma profundidade $\log^b n$ da raiz, visto que $n/b\ a_{\log}^{b}$ $_n$ = 1. Há $a^{\ logb}$ $_n$ $n_{\log}^{b\ a}$ folhas na árvore.

Podemos obter a equação (4.21) somando os custos dos nós em cada profundidade da árvore, como mostra a figura. O custo de todos os nós internos à profundidade j é a_j $f(n/b^{\ j}\ )$ e, assim, o total de todos os níveis de nós internos é

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j).$$

No algoritmo de divisão e conquista subjacente, essa soma representa os custos de dividir problemas em subproblemas e depois recombinar os subproblemas. O custo de todas as folhas, que é o custo de fazer com que todos os $n_{\log}^{b\ a}$ subproblemas tenham tamanho 1, é $\Theta(n_{\log}^{b\ a})$.

Em termos da árvore de recursão, os três casos do teorema mestre correspondem a casos nos quais o custo total da árvore é (1) dominado pelos custos nas folhas, (2) distribuído uniformemente entre os níveis da árvore ou (3) dominado pelo custo da raiz.

O somatório na equação (4.21) descreve o custo das etapas de divisão e combinação no algoritmo de divisão e conquista subjacente. O lema seguinte dá limites assintóticos para o crescimento do somatório.

Lema 4.3

Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida para potências exatas de b. Uma função $g(n)$ definida para potências exatas de b por

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{4.22}$$

tem os seguintes limites assintóticos para potências exatas de b:

1. Se $f(n) = O(n^{\log_b a-})$ para alguma constante > 0, então $g(n) = O(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, então $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. Se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e para todo n suficientemente grande, então $g(n) = \Theta(f(n))$.

Prova Para o caso 1, temos $f(n) = O(n_{\log}{}^{b\ a}-)$, o que implica que $f(n/b_j) = O((n/b_j)^{\log b\ a}-)$. Substituindo na equação (4.22) temos

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) \tag{4.23}$$

Limitamos o somatório dentro da notação O fatorando termos e simplificando, o que resulta em uma série geométrica crescente:

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(b^\epsilon\right)^j \\
&= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n}-1}{b^\epsilon-1}\right) \\
&= n^{\log_b a-\epsilon} \left(\frac{n^\epsilon-1}{b^\epsilon-1}\right).
\end{aligned}$$

Visto que b e são constantes, podemos reescrever a última expressão como $n_{\log}{}^{b\ a}$-$O(n) = O(n_{\log}{}^{b\ a})$. Substituindo o somatório na equação (4.8) por essa expressão, temos

$$g(n) = O(n^{\log_b a}),$$

e o caso 1 fica provado.

Como o caso 2 considera que $f(n) = \Theta\ (n_{\log}{}^{b\ a})$ para o caso 2, temos que $f(n/b_j) = \Theta\ ((n/b^j)^{\log b\ a})$. Substituindo na equação (4.22) temos

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right). \tag{4.24}$$

Limitamos o somatório dentro da notação Θ como no caso 1, mas dessa vez não obtemos uma série geométrica. Em vez disso, descobrimos que todos os termos do somatório são iguais:

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} = n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j$$
$$= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1$$
$$= n^{\log_b a} \log_b n.$$

Substituindo o somatório da equação (4.24) por essa expressão obtemos

$$g(n) = \Theta(n^{\log_b a} \log_b n)$$
$$= \Theta(n^{\log_b a} \lg n),$$

o que prova o caso 2.

Provamos o caso 3 de modo semelhante. Como $f(n)$ aparece na definição (4.22) de $g(n)$ e todos os termos de $g(n)$ são não negativos, podemos concluir que $g(n) = (f(n))$ para potências exatas de b. No enunciado do lema consideramos que $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todo n suficientemente grande. Reescrevemos essa expressão como $f(n/b) \leq (c/a) f(n)$ e iteramos j vezes, o que produz $f(n/b_j) \leq (c/a)^j f(n)$ ou, o que é equivalente, $a_j f(n/b_j) \leq c^j f(n)$, onde consideramos que os valores para os quais efetuamos a iteração são suficientemente grandes. Visto que o último e menor desses valores é n/b_j-1, basta para considerarmos que n/b_j-1 é suficientemente grande.

Substituindo na equação (4.22) e simplificando, obtemos uma série geométrica mas, ao contrário da série no caso 1, essa tem termos decrescentes. Usamos um termo $O(1)$ para capturar os termos que não são abrangidos pela premissa que adotamos, isto é, n é suficientemente grande:

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$
$$\leq \sum_{j=0}^{\log_b n-1} c^j f(n) + O(1)$$
$$\leq f(n) \sum_{j=0}^{\infty} c^j + O(1)$$
$$= f(n)\left(\frac{1}{1-c}\right) + O(1)$$
$$= O(f(n)).$$

visto que c é uma constante. Assim, podemos concluir que $g(n) = \Theta(f(n))$ para potências exatas de b. Com a prova do caso 3, concluímos a prova do lema.

Agora podemos provar uma versão do teorema mestre para o caso em que n é uma potência exata de b.

Lema 4.4

Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida para potências exatas de b. Defina $T(n)$ para potências exatas de b pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1, \\ aT(n/b) + f(n) & \text{se } n = b^i, \end{cases}$$

onde i é um inteiro positivo. Então, $T(n)$ tem os seguintes limites assintóticos para potências exatas de b:

1. Se $f(n) = O(n^{\log_b a -})$ para alguma constante > 0, então $T(n) = \Theta(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Se $f(n) = (n^{\log_b a +})$ para alguma constante > 0, e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todo n suficientemente grande, então $T(n) = \Theta(f(n))$.

Prova Empregamos os limites do Lema 4.3 para avaliar o somatório (4.21) do Lema 4.2. Para o caso 1, temos

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}), \end{aligned}$$

e, para o caso 2,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n). \end{aligned}$$

Para o caso 3,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)). \end{aligned}$$

porque $f(n) = \Omega(n^{\log_b a + \epsilon})$

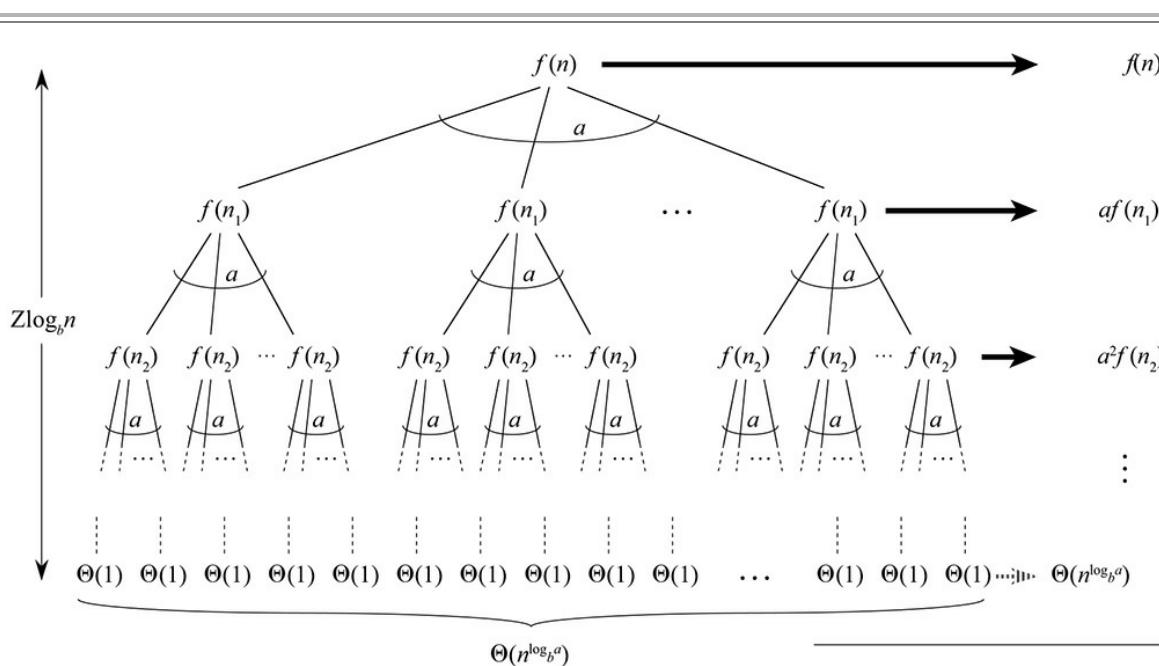
4.6.2 PISOS E TETOS

Para concluir a prova do teorema mestre, devemos agora estender nossa análise à situação na qual pisos e tetos aparecem na recorrência mestre, de modo que a recorrência é definida para todos os inteiros, não apenas para potências exatas de b. Obter um limite inferior para

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.25}$$

e um limite superior para

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.26}$$

é rotina, visto que podemos usar a limitação $n/b \geq n/b$ no primeiro caso para produzir o resultado desejado e usar a limitação $n/b \leq n/b$ no segundo caso. Para impor um limite inferior para a recorrência (4.26), utilizamos praticamente a mesma técnica usada para impor um limite superior para a recorrência (4.25); portanto, apresentaremos somente este último limite.

Modificamos a árvore de recursão da Figura 4.7 para produzir a árvore de recursão da Figura 4.8. À medida que descemos pela árvore de recursão, obtemos uma sequência de invocações recursivas para os argumentos

$$
\begin{aligned}
&n\,,\\
&\lceil n/b \rceil\,,\\
&\lceil \lceil n/b \rceil /b \rceil\,,\\
&\lceil \lceil \lceil n/b \rceil /b \rceil /b \rceil\,,\\
&\quad\vdots
\end{aligned}
$$

Vamos denotar o j-ésimo elemento na sequência por n_j, onde

$$
n_j = \begin{cases} n & \text{se } j = 0,\\ \lceil n_{j-1}/b \rceil & \text{se } j > 0. \end{cases} \tag{4.27}
$$

Nossa primeira meta é determinar a profundidade k tal que n_k é uma constante. Usando a desigualdade$x \leq x + 1$, obtemos

Figura 4.8 Árvore de recursão gerada por $T(n) = aT(n/b) + f(n)$. O argumento recursivo n_j é dado pela equação (4.27).

$$
\begin{aligned}
n_0 &\leq n, \\
n_1 &\leq \frac{n}{b}+1, \\
n_2 &\leq \frac{n}{b^2}+\frac{n}{b}+1, \\
n_3 &\leq \frac{n}{b^3}+\frac{n}{b^2}+\frac{n}{b}+1, \\
&\vdots
\end{aligned}
$$

Em geral, temos

$$
\begin{aligned}
n_j &\leq \frac{n}{b^j}+\sum_{i=0}^{j-1}\frac{1}{b^i} \\
&< \frac{n}{b^j}+\sum_{i=0}^{\infty}\frac{1}{b^i} \\
&= \frac{n}{b^j}+\frac{b}{b-1}.
\end{aligned}
$$

Fazendo $j = \lfloor \log_{b\,n} \rfloor$, obtemos

$$
\begin{aligned}
n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lfloor \log_b n \rfloor}}+\frac{b}{b-1} \\
&< \frac{n}{b^{\log_b n-1}}+\frac{b}{b-1} \\
&= \frac{n}{n/b}+\frac{b}{b-1} \\
&= b+\frac{b}{b-1} \\
&= O(1),
\end{aligned}
$$

e, assim, vemos que, à profundidade $\log_b n$, o tamanho do problema é limitado por uma constante.

Pela Figura 4.8, observamos que

$$
T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor -1} a^j f(n_j), \tag{4.28}
$$

que é quase igual à equação (4.21), exceto que n é um inteiro arbitrário e não está restrito a ser uma potência exata de b.

Agora podemos avaliar o somatório

$$
g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor -1} a^j f(n_j), \tag{4.29}
$$

pela equação (4.28) de modo análogo à prova do Lema 4.3. Começando com o caso 3, se $a\,f(n/b) \leq c\,f(n)$ para $n > b + b/(b - 1)$, onde $c < 1$ é uma constante, então segue-se que $a_j f(n\ ) \leq c\,j\,f(n)$. Portanto, podemos avaliar a soma na equação (4.29) exatamente como no Lema 4.3. Para o caso 2, temos $f(n) = \Theta(n_{\log^b a})$. Se pudermos mostrar que $f(n\ )$

$= O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, então a prova para o caso 2 do Lema 4.3 funcionará. Observe que $j \leq \log_b n$ implica $b^j/n \leq 1$. O limite $f(n) = O(n^{\log_b a})$ implica que existe uma constante $c > 0$ tal que, para todo n suficientemente grande,

$$
\begin{aligned}
f(n_j) &\leq c\left(\frac{n}{b^j}+\frac{b}{b-1}\right)^{\log_b a} \\
&= c\left(\frac{n}{b^j}\left(1+\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a} \\
&= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\left(\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a} \\
&= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\frac{b}{b-1}\right)^{\log_b a} \\
&= O\left(\frac{n^{\log_b a}}{a^j}\right),
\end{aligned}
$$

visto que $c(1 + b/(b - 1))^{\log_b a}$ é uma constante. Assim, provamos o caso 2. A prova do caso 1 é quase idêntica. A chave é provar o limite $f(n_j) = O(n^{\log_b a-e})$, que é semelhante à prova correspondente do caso 2, embora a álgebra seja mais complicada.

Agora provamos os limites superiores no teorema mestre para todos os inteiros n. A prova dos limites inferiores é semelhante.

Exercícios

4.6-1 ★

Dê uma expressão simples e exata para n_i na equação (4.27) para o caso em que b é um inteiro positivo, em vez de um número real arbitrário.

4.6-2 ★

Mostre que, se $f(n) = \Theta(n^{\log_b a} \lg^k n)$ onde $k \geq 0$, então a recorrência mestre tem solução $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Por simplicidade, restrinja sua análise a potências exatas de b.

4.6-3 ★

Mostre que o caso 3 do teorema mestre é exagerado, no sentido de que a condição de regularidade $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ implica que existe uma constante > 0 tal que $f(n) = (n^{\log_b a+})$.

Problemas

4-1 Exemplos de recorrência

Dê limites assintóticos superiores e inferiores para $T(n)$ em cada uma das recorrências a seguir. Considere que $T(n)$ é constante para $n \leq 2$. Torne seus limites tão restritos quanto possível e justifique suas respostas.

a. $T(n) = 2T(n/2) = n_4$.

b. $T(n) = T(7n/10) = n$.

c. $T(n) = 16T(n/4) = n_2$.

d. $T(n) = 7T(n/3) = n_2$.

e. $T(n) = 7T(n/2) = n_2$.

f. $T(n) = 2T(n/4) = \sqrt{n}$.

g. $T(n) = T(n - 2) = n_2$.

4-2 Custos da passagem de parâmetros

Em todo este livro, supomos que a passagem de parâmetros durante chamadas de procedimento demora um tempo constante, mesmo para passar um arranjo de N elementos. Essa premissa é válida na maioria dos sistemas porque é passado um ponteiro para o arranjo, e não o próprio arranjo. Este problema examina as implicações de três estratégias de passagem de parâmetros:

1. Um arranjo é passado por ponteiro. Tempo = $\Theta(1)$.

2. Um arranjo é passado por cópia. Tempo = $\Theta(N)$, onde N é o tamanho do arranjo.

3. Um arranjo é passado por cópia somente da subfaixa que poderia ser acessada pelo procedimento chamado. Tempo = $\Theta(q - p + 1)$ se o subarranjo $A[p .. q]$ for passado.

a. Considere o algoritmo de busca binária recursiva para localizar um número em um arranjo ordenado (veja Exercício 2.3-5). Dê recorrências para os tempos de execução do pior caso de busca binária quando os arranjos são passados com a utilização de cada um dos três métodos citados e dê bons limites superiores para as soluções das recorrências. Seja N o tamanho do problema original e n o tamanho de um subproblema.

b. Faça novamente a parte (a) para o algoritmo MERGE-SORT da Seção 2.3.1.

4-3 Outros exemplos de recorrência

Dê limites assintóticos superiores e inferiores para $T(n)$ em cada uma das recorrências a seguir. Considere que $T(n)$ é constante para n suficientemente pequeno. Torne seus limites tão justos quanto possível e justifique suas respostas.

a. $T(n) = 4T(n/3) + n \lg n$.

b. $T(n) = 3T(n/3) + n/\lg n$.

c. $T(n) = 4T(n/2) + n_2 \sqrt{n}$.

d. $T(n) = 3T(n/3 - 2) + n/2$.

e. $T(n) = 2T(n/2) + n/\lg n$.

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n - 1) + 1/n$.

h. $T(n) = T(n - 1) + \lg n$.

i. $T(n) = T(n - 2) + 1/\lg n$.

j. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4-4 Números de Fibonacci

Este problema desenvolve propriedades dos números de Fibonacci, que são definidos pela recorrência (3.22). Usaremos a técnica de gerar funções para resolver a recorrência de Fibonacci. Defina a ***função geradora*** (ou ***série formal de potências***) F por

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots,\end{aligned}$$

onde F_i é o i-ésimo número de Fibonacci.

a. Mostre que $F(z) = z + zF(z) + z_2F(z)$.

b. Mostre que

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1-z-z^2} \\ &= \frac{z}{(1-\phi z)(1-\hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z}\right).\end{aligned}$$

onde

$$\phi = \frac{1+\sqrt{5}}{2} = 1{,}61803\ldots$$

e

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0{,}61803\ldots$$

c. Mostre que

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^i - \hat{\phi}^i) z^i .$$

d. Use a parte (c) para provar que $F = \varphi_i / \sqrt{5}$ para $i > 0$, arredondado até o inteiro mais próximo. (*Sugestão*: Observe que $|\varphi\hat{}| < 1$.)

4-5 Testes de chips

O professor Diógenes tem n chips de circuito integrado supostamente idênticos que, em princípio, são capazes de testar uns aos outros. O aparelho de teste do professor acomoda dois chips de cada vez. Quando o aparelho é carregado, cada chip testa o outro e informa se este está bom ou ruim. Um chip bom sempre informa com precisão se o outro chip está bom ou ruim, mas o professor não pode confiar na resposta de um chip ruim. Portanto, os quatro resultados possíveis de um teste são:

Chip *A* informa	Chip *B* informa	Conclusão
B está bom	*A* está bom	Ambos estão bons ou ambos estão ruins
B está bom	*A* está ruim	Ao menos um está ruim
B está ruim	*A* está bom	Ao menos um está ruim
B está ruim	*A* está ruim	Ao menos um está ruim

a. Mostre que, se mais de $n/2$ chips estiverem ruins, o professor não pode necessariamente determinar quais chips estão bons usando qualquer estratégia baseada nessa espécie de teste aos pares. Admita que os chips ruins possam conspirar para enganar o professor.

b. Considere o problema de descobrir um único chip bom entre n chips, considerando que mais de $n/2$ dos chips estejam bons. Mostre que $n/2$ testes de pares são suficientes para reduzir o problema a um outro com aproximadamente metade do tamanho.

c. Mostre que os chips bons podem ser identificados com $\Theta(n)$ testes de pares, considerando que mais de $n/2$ dos chips estão bons. Dê e resolva a recorrência que descreve o número de testes.

1-7 ***Arranjos de Monge***

Um arranjo $m \times n$ de números reais, representado por A é um ***arranjo de Monge*** se, para todo i, j, k e l tais que $1 \le i < k \le m$ e $1 \le j < l \le n$, temos

$$A[i, j] + A[k, l] \le A[i, l] + A[k, j].$$

Em outras palavras, sempre que escolhemos duas linhas e duas colunas de um arranjo de Monge e consideramos os quatro elementos nas interseções das linhas e das colunas, a soma dos elementos na parte superior esquerda e na parte inferior direita é menor ou igual à soma dos elementos na parte inferior esquerda e na parte superior direita. Por exemplo, o arranjo a seguir é um arranjo de Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Prove que um arranjo é de Monge se e somente se para todo $i = 1, 2, ..., m - 1$ e $j = 1, 2, ..., n - 1$, temos

$$A[i, j] + A[i + 1, j + 1] \le A[i, j + 1] + A[i + 1, j].$$

(*Sugestão:* Para a parte "se", aplique indução às linhas e colunas separadamente.)

b. O arranjo a seguir não é de Monge. Troque a ordem de um elemento para transformá-lo em um arranjo de Monge. (*Sugestão:* Use a parte (a).)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. Seja $f(i)$ o índice da coluna que contém o elemento mínimo da extrema esquerda da linha i. Prove que $f(1) \le f(2) \le ... \le f(m)$ para qualquer arranjo de Monge $m \times n$.

d. Apresentamos a seguir a descrição de um algoritmo de divisão e conquista que calcula o elemento mínimo da extrema esquerda em cada linha de um arranjo de Monge $m \times n$ A:

Construa uma submatriz A' de A composta pelas linhas de numeração par de A. Determine recursivamente o mínimo da extrema esquerda para cada linha de A'. Em seguida, calcule o mínimo da extrema esquerda nas linhas de numeração ímpar de A.

Explique como calcular o mínimo da extrema esquerda nas linhas de numeração ímpar de A (dado que o mínimo da extrema esquerda das linhas de numeração par seja conhecido) no tempo $O(m + n)$.

e. Escreva a recorrência que descreve o tempo de execução do algoritmo descrito na parte (d). Mostre que sua solução é $O(m + n \log m)$.

NOTAS DO CAPÍTULO

O método de divisão e conquista como técnica para projeto de algoritmos data, no mínimo, de 1962, com a publicação de um artigo por Karatsuba e Ofman [194]. Todavia, é possível que ele tenha sido usado bem antes disso; de acordo com Heideman, Johnson e Burrus [163], C. F.Gauss inventou o primeiro algoritmo de transformada rápida de Fourier em 1805, e a formulação de Gauss desmembra o problema em subproblemas menores, cujas soluções são combinadas.

O problema do subarranjo mínimo na Seção 4.1 é uma pequena variação de um problema estudado por Bentley [43, Capítulo 7].

O algoritmo de Strassen [325] causou grande sensação quando foi publicado em 1969. Antes dessa data, poucos imaginavam a possibilidade de um algoritmo assintoticamente mais rápido do que o procedimento básico SQUARE-MATRIX-MULTIPLY. O limite superior assintótico para multiplicação de matrizes foi melhorado desde então. Até agora, o algoritmo mais assintoticamente eficiente para multiplicar matrizes $n \times n$, proposto por Coppersmith e Winograd [79], tem tempo de execução de $O(n_2,^{376})$. O melhor limite inferior conhecido é apenas o óbvio limite inferior (n_2) (óbvio porque temos de preencher n_2 elementos da matriz do produto).

De um ponto de vista prático, muitas vezes o algoritmo de Strassen não é o método de preferência para multiplicação de matrizes, por quatro razões:

1. O fator constante oculto no tempo de execução $\Theta(n_{\lg 7})$ do algoritmo de Strassen é maior do que o fator constante no tempo $\Theta(n_3)$ do procedimento SQUARE-MATRIX-MULTIPLY.
2. Quando as matrizes são esparsas, os métodos específicos para matrizes esparsas são mais rápidos.
3. O algoritmo de Strassen não é tão numericamente estável quanto o procedimento SQUARE--MATRIX-MULTIPLY. Em outras palavras, em razão da utilização de aritmética de computador de precisão limitada para tratar valores não inteiros, acumulam-se erros maiores no algoritmo de Strassen do que no procedimento SQUARE-MATRIX-MULTIPLY.
4. As submatrizes formadas nos níveis de recursão consomem espaço.

As duas últimas razões foram atenuadas por volta de 1990. Higham [167] demonstrou que a diferença em estabilidade numérica tinha sido alvo de excessiva ênfase; embora o algoritmo de Strassen seja demasiadamente instável numericamente para algumas aplicações, essa instabilidade está dentro de limites aceitáveis para outras. Bailey, Lee e Simon [32] discutem técnicas para reduzir os requisitos de memória para o algoritmo de Strassen.

Na prática, implementações de multiplicação rápida de matrizes para matrizes densas usam o algoritmo de Strassen para tamanhos de matrizes acima de um "ponto de passagem" e trocam para um método mais simples tão logo o tamanho do subproblema se reduza a uma dimensão abaixo do ponto de passagem. O valor exato do ponto de passagem depende muito do sistema. As análises que contam operações mas ignoram os efeitos de caches e pipelines produziram pontos de passagem baixos de até $n = 8$ (por Higham [167]) ou $n = 12$ (por Huss-Lederman *et al.* [186]). D'Alberto e Nicolau [81] desenvolveram um esquema adaptativo que determina o ponto de passagem por comparação com paradigmas estabelecidos quando da instalação de seu pacote de software. Eles encontraram pontos de passagem em vários sistemas na faixa de $n = 400$ a $n = 2150$ e não conseguiram determinar um ponto de passagem para alguns sistemas.

Recorrências já eram estudadas em 1202 por L. Fibonacci, e é a ele que os números de Fibonacci devem sua denominação. A. De Moivre introduziu o método de funções geradoras (veja Problema 4-4) para resolver recorrências. O método mestre foi adaptado de Bentley, Haken e Saxe [44], que fornece o método estendido justificado pelo Exercício 4.6-2. Knuth [209] e Liu [237] mostram como resolver recorrências lineares usando o método de funções

geradoras. Purdom e Brown [287] e Graham, Knuth e Patashnik [152] contêm discussões extensas da solução de recorrências.

Vários pesquisadores, entre eles Akra e Bazzi [13], Roura [299], Verma [346] e Yap [306], propuseram métodos para resolver recorrências de divisão e conquista mais gerais do que as que são resolvidas pelo método mestre. Descrevemos aqui o resultado de Akra e Bazzi , como modificado por Leighton (228). O método de Akra-Bazzi funciona para recorrências da forma

$$T(n)=\begin{cases}\Theta(1) & \text{se } 1\leq x\leq x_0,\\ \sum_{i=1}^{k} a_i T(b_i x)+f(x) & \text{se } x>x_0,\end{cases} \tag{4.30}$$

onde

- $x \geq 1$ é um número real,
- x_0 é uma constante tal que $x_0 \geq 1/b_i$ e $x_0 \geq 1/(1 - b_i)$ para $i = 1, 2, \dots, k$,
- a_i é uma constante positiva para $i = 1, 2, \dots, k$,
- b_i é uma constante na faixa $0 < b_i < 1$ para $i = 1, 2, ..., k$,
- $k \geq 1$ é uma constante inteira e
- $f(x)$ é uma função não negativa que satisfaz a ***condição de crescimento polinomial:*** existem constantes positivas c_1 e c_2 tais que, para todo $x \geq 1$, para $i = 1, 2, ..., k$, e para todo u tal que $b_i x \leq u \leq x$, temos $c_1 f(x) \leq f(u) \leq c_2 f(x)$. (Se $|f\,'(x)|$ tiver como limite superior algum polinômio em x, então $f(x)$ satisfaz a condição de crescimento polinomial. Por exemplo, $f(x) = x^a \lg^b x$ satisfaz essa condição para quaisquer constantes reais a e b.)

Embora o método mestre não se aplique a um recorrência como $T(n) = T(n/3) + T(2n/3) + O(n)$, o método de Akra-Bazzi se aplica. Para resolver a recorrência (4.30), em primeiro lugar determine o único número real p tal que $\sum_{i=1}^{k} a_i b_i^p = 1$ (Tal p sempre existe.) Então, a solução para a recorrência é

$$T(n)=\Theta\left(x^p\left(1+\int_1^x \frac{f(u)}{u^{p+1}}\right)\right).$$

O método de Akra-Bazzi pode ser um pouco difícil de usar, mas serve para resolver recorrências que modelam a divisão do problema em subproblemas de tamanhos substancialmente desiguais. O método mestre é mais simples de usar, mas só se aplica quando os tamanhos dos subproblemas são iguais.

5 ANÁLISE PROBABILÍSTICA E ALGORITMOS ALEATORIZADOS

Este capítulo introduz a análise probabilística e os algoritmos aleatorizados. Se você não estiver familiarizado com os fundamentos da teoria das probabilidades, leia o Apêndice C, que apresenta uma revisão desse assunto. A análise probabilística e os algoritmos aleatorizados serão revistos várias vezes ao longo deste livro.

5.1 O PROBLEMA DA CONTRATAÇÃO

Suponha que você precise contratar um novo auxiliar de escritório. Suas tentativas anteriores de contratação foram malsucedidas e você decidiu usar uma agência de empregos. A agência de empregos lhe envia um candidato por dia e você o entrevista e depois decide contratá-lo ou não. Você terá de pagar à agência de empregos uma pequena taxa para entrevistar um candidato. Porém, a contratação de um candidato é mais onerosa, já que você tem de demitir seu auxiliar de escritório atual e pagar uma taxa de contratação substancial à agência de empregos. A política de sua empresa é ter sempre a melhor pessoa possível para o cargo. Portanto, você decide que, depois de entrevistar cada candidato, se esse candidato for mais bem qualificado que o auxiliar de escritório atual, o auxiliar de escritório atual será demitido e o novo candidato será contratado. Você está disposto a pagar o preço resultante dessa estratégia, mas deseja avaliar qual será esse preço.

O procedimento HIRE-ASSISTANT dado a seguir expressa essa estratégia de contratação em pseudocódigo. Tal procedimento considera que os candidatos ao emprego de auxiliar de escritório são numerados de 1 a n e também que, depois de entrevistar o candidato i, você poderá determinar se esse candidato i é o melhor que viu até então. Para inicializar, o procedimento cria um candidato fictício, de número 0, menos qualificado que cada um dos outros candidatos.

```
HIRE-ASSISTANT (n)
1    melhor = 0       // o candidato 0 é um candidato fictício menos qualificado
2    for i = 1 to n
3         entrevistar candidato i
4         if candidato i é melhor que candidato melhor
5              melhor = i
6              contratar candidato i
```

O modelo de custo para esse problema é diferente do modelo descrito no Capítulo 2. Não estamos preocupados com o tempo de execução de HIRE-ASSISTANT, mas com os custos incorridos na entrevista e na contratação. À primeira vista, analisar o custo desse algoritmo pode parecer muito diferente de analisar o tempo de execução, digamos, da ordenação por intercalação. Porém, as técnicas analíticas usadas são idênticas, quer estejamos analisando custo ou

tempo de execução. Em um ou outro caso, estamos contando o número de vezes que certas operações básicas são executadas.

Entrevistar tem um custo baixo, digamos c_i, enquanto contratar é caro e custa c_h. Se m for o número de pessoas contratadas, o custo total associado a esse algoritmo é $O(c_i n + c_h m)$. Independentemente de quantas pessoas contratarmos, sempre entrevistaremos n candidatos e, portanto, sempre incorreremos no custo $c_i n$ associado a entrevistar. Assim, nos concentraremos na análise de $c_h m$, o custo de contratação. Essa quantidade varia com cada execução do algoritmo.

Esse cenário serve como modelo para um paradigma computacional bem comum. Muitas vezes, precisamos determinar o valor máximo ou mínimo em uma sequência examinando cada elemento da sequência e mantendo um "vencedor" atual. O problema da contratação modela a frequência com que atualizamos nossa noção de qual elemento está vencendo no momento em questão.

Análise do pior caso

No pior caso, contratamos cada candidato que entrevistamos. Essa situação ocorre se os candidatos vierem em ordem estritamente crescente de qualidade, e nesse caso contratamos n vezes para um custo total de contratação $O(c_h n)$.

Porém, é claro que os candidatos nem sempre vêm em ordem crescente de qualidade. De fato, não temos nenhuma ideia da ordem em que eles chegam nem temos qualquer controle sobre essa ordem. Portanto, é natural perguntar o que esperamos que aconteça em um caso típico ou médio.

Análise probabilística

A ***análise probabilística*** é a utilização da probabilidade na análise de problemas. Na maior parte das vezes, usamos análise probabilística para analisar o tempo de execução de um algoritmo. Às vezes, nós a usamos para analisar outras quantidades, como o custo da contratação no procedimento HIRE-ASSISTANT. Para efetuar uma análise probabilística, temos de conhecer ou supor a distribuição das entradas. Em seguida, analisamos nosso algoritmo, calculando o tempo de execução para o caso médio, e tomamos a média sobre a distribuição das entradas possíveis. Assim, na verdade estamos calculando a média do tempo de execução de todas as entradas possíveis. Quando informarmos esse tempo de execução, nós o denominaremos ***tempo de execução do caso médio***.

Temos de tomar muito cuidado quando decidirmos a distribuição das entradas. Em alguns problemas é possível inferir razoavelmente alguma coisa sobre o conjunto de todas as entradas possíveis; então poderemos usar a análise probabilística como técnica para projetar um algoritmo eficiente e como um meio de compreender melhor um problema. Em outros problemas, não é possível descrever uma distribuição de entradas razoável e, nesses casos, não podemos utilizar a análise probabilística.

No caso do problema da contratação, podemos considerar que os candidatos vêm em uma ordem aleatória. O que isso significa para esse problema? Supomos que podemos comparar dois candidatos quaisquer e decidir qual é o mais bem qualificado; isto é, existe uma ordem total para os candidatos. (Consulte o Apêndice B para ver a definição de uma ordem total.) Assim, podemos classificar cada candidato com um número exclusivo de 1 a n, usando *ordenação*(i) para denotar o posto do candidato i, adotando a convenção de que um posto mais alto corresponde a um candidato mais bem qualificado. A lista ordenada $\langle$*ordenação*(1), *ordenação*(2), ..., *ordenação*(n)$\rangle$ é uma permutação da lista $\langle 1, 2, ..., n\rangle$. Dizer que os candidatos chegam em ordem aleatória equivale a dizer que essa lista de classificações tem igual probabilidade de ser qualquer uma das $n!$ permutações dos números 1 a n. Alternativamente, dizemos que as classificações formam uma ***permutação aleatória uniforme***; isto é, cada uma das $n!$ permutações possíveis aparece com igual probabilidade.

A Seção 5.2 contém uma análise probabilística do problema da contratação.

Algoritmos aleatorizados

Para utilizar a análise probabilística, precisamos saber alguma coisa sobre a distribuição das entradas. Em muitos casos, sabemos bem pouco sobre tal distribuição. Mesmo que saibamos algo sobre a distribuição, talvez não possamos modelar esse conhecimento em termos computacionais. Ainda assim, muitas vezes podemos usar probabilidade e aleatoriedade como ferramentas para projeto e análise de algoritmos, aleatorizando o comportamento de parte do algoritmo.

No problema da contratação, pode parecer que os candidatos estão sendo apresentados em ordem aleatória, mas não temos nenhum meio de saber se isso realmente acontece ou não. Portanto, para desenvolver um algoritmo aleatorizado para o problema da contratação, devemos ter maior controle sobre a ordem em que entrevistamos os candidatos. Portanto, vamos alterar um pouco o modelo. Dizemos que a agência de empregos tem n candidatos e que ela nos envia uma lista dos candidatos com antecedência. A cada dia, escolhemos aleatoriamente qual candidato entrevistar. Embora não saibamos nada sobre os candidatos (além de seus nomes), fizemos uma mudança significativa. Em vez de confiar em uma suposição de que os candidatos virão em ordem aleatória, obtivemos o controle do processo e impusemos uma ordem aleatória.

De modo mais geral, dizemos que um algoritmo é ***aleatorizado*** se seu comportamento for determinado não apenas por sua entrada, mas também por valores produzidos por um ***gerador de números aleatórios***. Consideraremos que temos à nossa disposição um gerador de números aleatórios RANDOM. Uma chamada a RANDOM(a, b) retorna um inteiro entre a e b, inclusive, sendo cada inteiro igualmente provável. Por exemplo, RANDOM(0, 1) produz 0 com probabilidade 1/2 e produz 1 com probabilidade 1/2. Uma chamada a RANDOM(3, 7) retorna 3, 4, 5, 6 ou 7, cada um com probabilidade 1/5. Cada inteiro retornado por RANDOM é independente dos inteiros retornados em chamadas anteriores. Você pode imaginar RANDOM como o lançamento de um dado de $(b - a + 1)$ lados para obter sua saída. (Na prática, a maioria dos ambientes de programação oferece um ***gerador de números pseudoaleatórios***: um algoritmo determinístico que retorna números que "parecem" aleatórios estatisticamente.)

Quando analisamos o tempo de execução de um algoritmo aleatorizado, adotamos a expectativa do tempo de execução para a distribuição de valores retornada pelo gerador de números aleatórios. Distinguimos esses algoritmos daqueles, cuja entrada é aleatória referindo-nos ao tempo de execução de um algoritmo aleatorizado como um ***tempo de execução esperado.*** Em geral, discutimos o tempo de execução do caso médio quando a distribuição de probabilidade refere-se às entradas do algoritmo, e discutimos o tempo de execução esperado quando o próprio algoritmo faz escolhas aleatórias.

Exercícios

5.1-1 Mostre que a suposição de que sempre somos capazes de determinar qual candidato é o melhor, na linha 4 do procedimento HIRE-ASSISTANT, implica que conhecemos uma ordem total para as classificações dos candidatos.

5.1-2 ★

Descreva uma implementação do procedimento RANDOM(a, b) que só faça chamadas a RAN- DOM(0, 1). Qual é o tempo de execução esperado de seu procedimento, em função de a e b?

5.1-3 ★

Suponha que você queira que saia 0 com probabilidade 1/2 e 1 com probabilidade 1/2. Há um procedimento BIASED-RANDOM à sua disposição que produz como saída 0 ou 1. A saída é 1 com alguma probabilidade p e 0 com probabilidade $1 - p$, onde $0 < p < 1$, mas você não sabe qual é o valor de p. Dê um algoritmo que utilize BIASED-RANDOM como uma sub-rotina e retorne uma resposta não enviesada, retornando 0 com probabilidade 1/2 e 1 com probabilidade 1/2. Qual é o tempo de execução esperado de seu algoritmo em função de p?

5.2 VARIÁVEIS ALEATÓRIAS INDICADORAS

Para analisar muitos algoritmos, inclusive o problema da contratação, usamos variáveis aleatórias indicadoras. Variáveis aleatórias indicadoras nos dão um método conveniente para converter probabilidades em expectativas. Suponha que temos um espaço amostral S e um evento A. Então, a ***variável aleatória indicadora*** $\text{I}\{A\}$ associada ao evento A é definida como

$$\text{I}\{A\} = \begin{cases} 1 & \text{se } A \text{ ocorrer,} \\ 0 & \text{se } A \text{ não ocorrer.} \end{cases} \tag{5.1}$$

Como um exemplo simples, vamos determinar o número esperado de caras que obtemos quando lançamos uma moeda não viciada. Nosso espaço amostral é $S = \{H, T\}$ com $\Pr\{H\} = \Pr\{T\} = 1/2$. Então, podemos definir uma variável aleatória indicadora X_H, associada ao resultado "cara" do lançamento da moeda, que é o evento H. Essa variável conta o número de caras obtidas nesse lançamento, e é 1 se a moeda der cara, caso contrário é 0. Escrevemos

$$\begin{aligned} X_H &= \text{I}\{H\} \\ &= \begin{cases} 1 & \text{se } H \text{ ocorrer,} \\ 0 & \text{se } T \text{ ocorrer.} \end{cases} \end{aligned}$$

O número esperado de caras obtidas em um lançamento da moeda é simplesmente o valor esperado de nossa variável indicadora X_H:

$$\begin{aligned} \text{E}[X_H] &= \text{E}[\text{I}\{H\}] \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 \,. \end{aligned}$$

Desse modo, o número esperado de caras obtidas por um lançamento de uma moeda não viciada é 1/2. Como mostra o lema a seguir, o valor esperado de uma variável aleatória indicadora associada a um evento A é igual à probabilidade de A ocorrer.

Lema 5.1

Dado um espaço amostral S e um evento A no espaço amostral S, seja $X_A = \text{I}\{A\}$. Então, $\text{E}\{X_A\} = \Pr\{A\}$. ***Prova*** Pela definição de variável aleatória indicadora da equação (5.1) e pela definição de valor esperado, temos

$$\begin{aligned} \text{E}[X_A] &= \text{E}[\text{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\bar{A}\} \\ &= \Pr\{A\} \,, \end{aligned}$$

onde $\bar{A}$ denota $S - A$, o complemento de A.

Embora variáveis aleatórias indicadoras possam parecer incômodas para uma aplicação como a contagem do número esperado de caras no lançamento de uma única moeda, elas são úteis para analisar situações em que realizamos testes aleatórios repetidos. Por exemplo, as variáveis aleatórias indicadoras nos dão um caminho simples para chegar ao resultado da equação (C.37). Nessa equação, calculamos o número de caras em n lançamentos da moeda, considerando separadamente a probabilidade de obter 0 cara, 1 cara, 2 caras etc. Ao contrário, o método mais simples proposto na equação (C.38) utiliza implicitamente variáveis aleatórias indicadoras. Tornando esse argumento mais explícito, fazemos X_i a variável aleatória indicadora associada ao evento no qual o i-ésimo lançamento dá cara: $X_i = \text{I}$ {o

i-ésimo lançamento resulta no evento $H\}$. Seja X a variável aleatória que denota o número total de caras nos n lançamentos da moeda, de modo que

$$X = \sum_{i=1}^{n} X_i.$$

Desejamos calcular o número esperado de caras; para isso, tomamos a esperança de ambos os lados da equação anterior para obter

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right].$$

Essa equação dá a esperança da soma de n variáveis aleatórias indicadoras. Pelo Lema 5.1, podemos calcular facilmente a esperança de cada uma das variáveis aleatórias. Pela equação (C.21) — linearidade de esperança — é fácil calcular a esperança da soma: ela é igual à soma das esperanças das n variáveis aleatórias. A linearidade de esperança torna a utilização das variáveis aleatórias indicadoras uma técnica analítica poderosa; ela se aplica até mesmo quando existe dependência entre as variáveis aleatórias. Agora é fácil calcular o número esperado de caras:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} 1/2 \\ &= n/2. \end{aligned}$$

Assim, em comparação com o método empregado na equação (C.37), as variáveis aleatórias indicadoras simplificam muito o cálculo. Utilizaremos variáveis aleatórias indicadoras em todo este livro.

Análise do problema da contratação com a utilização de variáveis aleatórias indicadoras

Voltando ao problema da contratação, agora desejamos calcular o número esperado de vezes que contratamos um novo auxiliar de escritório. Para usar uma análise probabilística, supomos que os candidatos chegam em ordem aleatória, como discutimos na seção anterior. (Veremos na Seção 5.3 como descartar essa premissa.) Seja X a variável aleatória cujo valor é igual ao número de vezes que contratamos um novo auxiliar de escritório. Então, poderemos aplicar a definição de valor esperado da equação (C.20) para obter

$$E[X] = \sum_{i=1}^{n} x \Pr\{X = x\},$$

mas esse cálculo seria incômodo. Em vez disso, utilizaremos variáveis aleatórias indicadoras para simplificar bastante o cálculo.

Para usar variáveis aleatórias indicadoras, em vez de calcular E[X] definindo uma única variável associada ao número de vezes que contratamos um novo auxiliar de escritório, definimos n variáveis relacionadas com a contratação ou não contratação de cada candidato específico. Em particular, tomamos X_i como a variável aleatória indicadora associada ao evento em que o i-ésimo candidato é contratado. Desse modo,

$$
\begin{aligned}
X_i &= \mathrm{I}\{\text{candidato i é contratado}\} \\
&= \begin{cases} 1 & \text{se candidato } i \text{ é contratado,} \\ 0 & \text{se candidato } i \text{ não é contratado,} \end{cases}
\end{aligned}
$$

e

$$X = X_1 + X_2 + \cdots + X_n \tag{5.2}$$

Pelo Lema 5.1, temos que

$\mathrm{E}[X_i] = \Pr$ {o candidato i é contratado} ,

e, portanto, devemos calcular a probabilidade de as linhas 5-6 de HIRE-ASSISTANT serem executadas.

O candidato i é contratado, na linha 6, exatamente quando ele é melhor que cada um dos candidatos 1 a $i - 1$. Como supomos que os candidatos chegam em ordem aleatória, os primeiros i candidatos apareceram em ordem aleatória. Qualquer um desses i primeiros candidatos tem igual probabilidade de ser o mais bem qualificado até o momento. O candidato i tem uma probabilidade $1/i$ de ser mais bem qualificado que os candidatos 1 a $i - 1$ e, assim, uma probabilidade $1/i$ de ser contratado. Pela Lema 5.1, concluímos que

$$\mathrm{E}[X_i] = 1/i\,. \tag{5.3}$$

Agora podemos calcular E[X]:

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(pela equação (5.2))} \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(por linearidade da expectativa)} \\
&= \sum_{i=1}^{n} 1/i && \text{(pela equação (5.3))} \\
&= \ln n + O(1) && \text{(pela equação A.7)).}
\end{aligned}
$$

Apesar de entrevistarmos n pessoas, na realidade só contratamos aproximadamente $\ln n$ delas, em média. Resumimos esse resultado no lema a seguir.

Lema 5.2

Considerando que os candidatos sejam apresentados em ordem aleatória, o algoritmo HIRE-ASSISTANT tem um custo total de contratação $O(c_h \ln n)$ no caso médio.

Prova O limite decorre imediatamente de nossa definição do custo de contratação e da equação (5.5), que mostra que o número esperado de contratações é aproximadamente ln n.

O custo de contratação do caso médio é uma melhoria significativa em relação ao custo de contratação do pior caso, $O(n\, c_h)$.

Exercícios

5.2-1 Em HIRE-ASSISTANT, supondo que os candidatos sejam apresentados em ordem aleatória, qual é a probabilidade de você contratar exatamente uma vez? Qual é a probabilidade de você contratar exatamente n vezes?

5.2-2 Em HIRE-ASSISTANT, supondo que os candidatos sejam apresentados em ordem aleatória, qual é a probabilidade de você contratar exatamente duas vezes?

5.2-3 Use variáveis aleatórias indicadoras para calcular o valor esperado da soma de n dados.

5.2-4 Use variáveis aleatórias indicadoras para resolver o problema a seguir, conhecido como ***problema da chapelaria***. Cada um dos n clientes entrega um chapéu ao funcionário da chapelaria em um restaurante. O funcionário devolve os chapéus aos clientes em ordem aleatória. Qual é o número esperado de clientes que recebem de volta seus próprios chapéus?

5.2-5 Seja $A[1 \,..\, n]$ um arranjo de n números distintos. Se $i < j$ e $A[i] > A[j]$, então o par (i, j) é denominado ***inversão*** de A. (Veja no Problema 2-4 mais informações sobre inversões.) Suponha que os elementos de A formem uma permutação aleatória uniforme de $\langle 1, 2, \ldots, n \rangle$. Use variáveis aleatórias indicadoras para calcular o número esperado de inversões.

5.3 ALGORITMOS ALEATORIZADOS

Na seção anterior, mostramos como conhecer uma distribuição para as entradas pode nos ajudar a analisar o comportamento do caso médio de um algoritmo. Muitas vezes, não temos tal conhecimento, o que impossibilita uma análise do caso médio. Como mencionamos na Seção 5.1, talvez possamos usar um algoritmo aleatorizado.

No caso de um problema como o da contratação, no qual é útil considerar que todas as permutações da entrada são igualmente prováveis, uma análise probabilística pode orientar o desenvolvimento de um algoritmo aleatorizado. Em vez de supor uma distribuição de entradas, impomos uma distribuição. Em particular, antes de executar o algoritmo, permutamos aleatoriamente os candidatos, de modo a impor a propriedade de cada permutação ser igualmente provável. Embora tenhamos modificado o algoritmo, ainda esperamos contratar um novo auxiliar de escritório aproximadamente ln n vezes. Porém, agora esperamos que seja esse o caso para *qualquer* entrada, e não somente para entradas obtidas de uma distribuição particular.

Vamos explorar um pouco mais a distinção entre análise probabilística e algoritmos aleatorizados. Na Seção 5.2 afirmamos que, supondo que os candidatos se apresentem em ordem aleatória, o número esperado de vezes que contratamos um novo auxiliar de escritório é aproximadamente ln n. Observe que aqui o algoritmo é determinístico; para qualquer entrada particular, o número de vezes que contratamos um novo auxiliar de escritório é sempre o mesmo. Além disso, o número de vezes que contratamos um novo auxiliar de escritório é diferente para entradas diferentes e depende das classificações dos diversos candidatos. Visto que esse número depende apenas das classificações dos candidatos, podemos representar uma entrada particular fazendo uma lista ordenada das classificações dos candidatos,

isto é, ⟨*ordenação*(1), *ordenação*(2), ..., *ordenação*(n)⟩. Dada a lista de classificações A_1 = ⟨1, 2, 3, 4, 5, 6, 7, 8, 9, 10⟩, um novo auxiliar de escritório é sempre contratado 10 vezes, já que cada candidato sucessivo é melhor que o anterior, e as linhas 5-6 são executadas em cada iteração. Dada a lista de classificações A_2 = ⟨10, 9, 8, 7, 6, 5, 4, 3, 2, 1⟩, um novo auxiliar de escritório é contratado apenas uma vez, na primeira iteração. Dada uma lista de classificações A_3 = ⟨5, 2, 1, 8, 4, 7, 10, 9, 3, 6⟩, um novo auxiliar de escritório é contratado três vezes após as entrevistas com os candidatos classificados como 5, 8 e 10. Lembrando que o custo de nosso algoritmo depende de quantas vezes contratamos um novo auxiliar de escritório, vemos que existem entradas custosas, como A_1, entradas econômicas, como A_2, e entradas moderadamente custosas, como A_3.

Por outro lado, considere o algoritmo aleatorizado que primeiro permuta os candidatos e depois determina o melhor candidato. Nesse caso, a aleatoriedade está no algoritmo, e não na distribuição de entradas. Dada uma entrada específica, digamos a entrada A_3 citada, não podemos dizer quantas vezes o máximo será atualizado porque essa quantidade é diferente a cada execução do algoritmo. A primeira vez em que executamos o algoritmo sobre A_3 ele pode produzir a permutação A_1 e executar 10 atualizações; porém, na segunda vez em que executamos o algoritmo, podemos produzir a permutação A_2 e executar apenas uma atualização. Na terceira vez em que o executamos, podemos produzir algum outro número de atualizações. A cada vez que executamos o algoritmo, a execução depende das escolhas aleatórias feitas, e ela provavelmente será diferente da execução anterior do algoritmo. Para esse algoritmo e muitos outros algoritmos aleatorizados, *nenhuma entrada específica induz seu comportamento do pior caso*. Nem mesmo o seu pior inimigo poderá produzir um arranjo de entrada ruim, já que a permutação aleatória torna irrelevante a ordem de entrada. O algoritmo aleatorizado só funcional mal se o gerador de números aleatórios produzir uma permutação "azarada".

No caso do problema da contratação, a única alteração necessária no código é a permutação aleatória do arranjo.

```
RANDOMIZED-HIRE-ASISTANT(n)
1    permutar aleatoriamente a lista de candidatos
2    melhor = 0       // o  candidato 0 é um candidato fictício menos qualificado
3    for i = 1 to n
4         entrevistar candidato i
5         if candidato i é melhor que candidato melhor
6              melhor = i
7              contratar candidato i
```

Com essa mudança simples, criamos um algoritmo aleatorizado cujo desempenho corresponde àquele obtido considerando que os candidatos se apresentavam em ordem aleatória.

Lema 5.3

O custo de contratação esperado do procedimento RANDOMIZED-HIRE-ASSISTANT é $O(c_h \ln n)$.

Prova Depois de permutar o arranjo de entrada, chegamos a uma situação idêntica à da análise probabilística de HIRE-ASSISTANT.

A comparação entre os Lemas 5.2 e 5.3 destaca a diferença entre análise probabilística e algoritmos aleatorizados. No Lema 5.2, fazemos uma suposição sobre a entrada. No Lema 5.3, não fazemos tal suposição, embora aleatorizar a entrada demore algum tempo adicional. Para manter a consistência com a terminologia que adotamos, expressamos o Lema 5.2 em termos do custo de contratação do caso médio e o Lema 5.3 em termos do custo de contratação esperado. No restante desta seção, discutiremos algumas questões relacionadas com a permutação aleatória das entradas.

Permutação aleatória de arranjos

Muitos algoritmos aleatorizados aleatorizam a entrada permutando o arranjo de entrada dado. (Existem outras maneiras de usar aleatorização.) Aqui, discutiremos dois métodos para esse fim. Supomos que temos um arranjo A que, sem perda de generalidade, contém os elementos 1 a n. Nossa meta é produzir uma permutação aleatória do arranjo.

Um método comum é atribuir a cada elemento $A[i]$ do arranjo uma prioridade aleatória $P[i]$ e depois ordenar os elementos de A de acordo com essas prioridades. Por exemplo, se nosso arranjo inicial for $A = \langle 1, 2, 3, 4\rangle$ e escolhermos as prioridades aleatórias $P = \langle 36, 3, 62, 19\rangle$, produziremos um arranjo $B = \langle 2, 4, 1, 3\rangle$, já que a segunda prioridade é a menor, seguida pela quarta, depois pela primeira e finalmente pela terceira. Denominamos esse procedimento PERMUTE-BY-SORTING:

```
PERMUTE-BY-SORTING(A)
1    n = A . comprimento
2    seja P[1 : : n] um novo arranjo
3    for i = 1 to n
4         P[i] = RANDOM(1, n³)
5    ordenar A, usando P como chaves de ordenação
```

A linha 4 escolhe um número aleatório entre 1 e n_3. Usamos uma faixa de 1 a n_3 para que seja provável que todas as prioridades em P sejam únicas. (O Exercício 5.3-5 pede que você prove que a probabilidade de todas as entradas serem únicas é, no mínimo, $1 - 1/n$, e o Exercício 5.3-6 pergunta como implementar o algoritmo ainda que duas ou mais prioridades sejam idênticas.) Vamos supor que todas as prioridades são únicas.

A etapa demorada nesse procedimento é a ordenação na linha 5. Como veremos no Capítulo 8, se usarmos uma ordenação por comparação, a ordenação demorará o tempo (n lg n). Podemos atingir esse limite inferior, já que vimos que a ordenação por intercalação demora o tempo Q(n lg n). (Veremos na Parte II outras ordenações por comparação que tomam o tempo Q(n lg n). O Exercício 8.3-4 pede que você resolva o problema muito semelhante de ordenar números na faixa 0 a $n_3 - 1$ no tempo $O(n)$.) Depois da ordenação, se $P[i]$ for a j-ésima menor prioridade, então $A[i]$ encontra-se na posição j da saída. Dessa maneira, obtemos uma permutação. Resta provar que o procedimento produz uma ***permutação aleatória uniforme***, isto é, que a probabilidade de ele produzir cada permutação dos números 1 a n é a mesma.

Lema 5.4

O procedimento PERMUTE-BY-SORTING produz uma permutação aleatória uniforme da entrada, admitindo que todas as prioridades são distintas.

Prova Começamos considerando a permutação particular na qual cada elemento $A[i]$ recebe a i-ésima menor prioridade. Mostraremos que essa permutação ocorre com probabilidade de exatamente $1/n!$. Para $i = 1, 2, ..., n$, seja E_i o evento em que o elemento $A[i]$ recebe a i-ésima menor prioridade. Então, desejamos calcular a probabilidade de que, para todo i, ocorre o evento E_i, que é

$$\Pr\{E_1 \cap E_2 \cap E_3 \cap \cdots \cap E_{n-1} \cap E_n\}.$$

Usando o Exercício C.2-5, essa probabilidade é igual a

$$\Pr\{E_1\} \cdot \Pr\{E_2 \mid E_1\} \cdot \Pr\{E_3 \mid E_2 \cap E_1\} \cdot \Pr\{E_4 \mid E_3 \cap E_2 \cap E_1\} \\ \cdots \Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap \cdots \cap E_1\} \cdots \Pr\{E_n \mid E_{n-1} \cap \cdots \cap E_1\}.$$

Temos que $\Pr\{E_1\} = 1/n$ porque essa é a probabilidade de uma prioridade escolhida aleatoriamente em um conjunto de n ser a menor prioridade. Em seguida, observamos que $\Pr\{E_2 \mid E_1\} = 1/(n - 1)$ porque, dado que o elemento $A[1]$ tem a menor prioridade, cada um dos $n - 1$ elementos restantes tem igual chance de ter a segunda menor prioridade.

Em geral, para $i = 2, 3, ..., n$, temos que $\Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap \ldots \cap E_1\} = 1/(n - i + 1)$, visto que, dado que os elementos $A[1]$ até $A[i - 1]$ têm as $i - 1$ menores prioridades (em ordem), cada um dos $n - (i - 1)$ elementos restantes tem igual chance de ter a i-ésima menor prioridade. Assim, temos

$$\begin{aligned}\Pr\{E_1 \cap E_2 \cap E_3 \cap \cdots \cap E_{n-1} \cap E_n\} &= \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right) \\ &= \frac{1}{n!},\end{aligned}$$

e mostramos que a probabilidade de obter a permutação identidade é $1/n!$.
Podemos estender essa prova a qualquer permutação de prioridades. Considere uma permutação fixa qualquer $s = \langle s(1), s(2), ..., s(n)\rangle$ do conjunto $\{1, 2, ..., n\}$. Vamos denotar por r_i a classificação da prioridade atribuída ao elemento $A[i]$, onde o elemento com a j-ésima menor prioridade tem a classificação j. Se definirmos E_i como o evento no qual o elemento $A[i]$ recebe a $s(i)$-ésima menor prioridade ou $r_i = s(i)$, a mesma prova ainda se aplica. Portanto, se calcularmos a probabilidade de obter qualquer permutação específica, o cálculo será idêntico ao apresentado antes, de modo que a probabilidade de obter essa permutação também será $1/n!$.

Você poderia pensar que, para provar que uma permutação é uma permutação aleatória uniforme, é suficiente mostrar que, para cada elemento $A[i]$, a probabilidade de ele terminar na posição j é $1/n$. O Exercício 5.3-4 mostra que essa condição mais fraca é, de fato, insuficiente.

Um método melhor para gerar uma permutação aleatória é permutar o arranjo dado no lugar. O procedimento RANDOMIZE-IN-PLACE faz isso no tempo $O(n)$. Em sua i-ésima iteração, o procedimento escolhe o elemento $A[i]$ aleatoriamente entre os elementos $A[i]$ e $A[n]$. Após a i-ésima iteração, $A[i]$ nunca é alterado.

```
RANDOMIZE-IN-PLACE(A)
1   n = A.comprimento
2   for i = 1 to n
3         trocar A[i] com A[RANDOM(i, n)]
```

Usaremos um invariante de laço para mostrar que o procedimento RANDOMIZE-IN-PLACE produz uma permutação aleatória uniforme. Uma ***k*-permutação** sobre um conjunto de n elementos, é uma sequência que contém k dos n elementos, sem nenhuma repetição. (Consulte o Apêndice C.) Há $n!/(n - k)!$ dessas permutações k possíveis.

Lema 5.5

O procedimento RANDOMIZE-IN-PLACE calcula uma permutação aleatória uniforme.

Prova Usamos o seguinte invariante de laço:

> Imediatamente antes da i-ésima iteração do laço **for** das linhas 2-3, para cada $(i - 1)$-permutação possível dos n elementos, o subarranjo $A[1 .. i - 1]$ contém essa $(i - 1)$-permutação com probabilidade $(n - i + 1)!/n!$.

Precisamos mostrar que esse invariante é verdadeiro antes da primeira iteração do laço, que cada iteração do laço mantém o invariante e que o invariante fornece uma propriedade útil para mostrar correção quando o laço termina.

Inicialização: Considere a situação imediatamente antes da primeira iteração do laço, de modo que $i = 1$. O invariante de laço diz que, para cada 0-permutação possível, o subarranjo $A[1 .. 0]$ contém essa 0-permutação com probabilidade $(n - i + 1)!/n! = n!/n! = 1$. O subarranjo $A[1 .. 0]$ é um subarranjo vazio, e uma 0-permutação não tem nenhum elemento. Assim, $A[1 .. 0]$ contém qualquer 0-permutação com probabilidade 1, e o invariante de laço é válido antes da primeira iteração.

Manutenção: Supomos que, imediatamente antes da i-ésima iteração, cada $(i - 1) -$ permutação possível aparece no subarranjo $A[1 \mathrel{..} i - 1]$ com probabilidade $(n - i + 1)!/n!$, e mostraremos que, após a i-ésima iteração, cada i-permutação possível aparece no subarranjo $A[1 \mathrel{..} i]$ com probabilidade $(n - i)!/n!$. Então, incrementar i para a próxima iteração mantém o invariante de laço. Vamos examinar a i-ésima iteração. Considere uma i-permutação específica e denote os elementos que ela contém por $\langle x_1, x_2, ..., x_i \rangle$. Essa permutação consiste em uma $(i - 1)$-permutação $\langle x_1, ..., x_{i-1} \rangle$ seguida pelo valor x_i que o algoritmo insere em $A[i]$. Seja E_1 o evento no qual as primeiras $i - 1$ iterações criaram a $(i - 1)$-permutação $\langle x_1, ..., x_{i-1} \rangle$ específica em $A[1 \mathrel{..} i - 1]$. Pelo invariante de laço, $\Pr\{E_1\} = (n - i + 1)!/n!$. Seja E_2 o evento no qual a i-ésima iteração insere x_i na posição $A[i]$. A i-permutação $\langle x_1, ..., x_i \rangle$ aparece em $A[1 \mathrel{..} i]$ exatamente quando ocorrem E_1 e E_2, e assim desejamos calcular $\Pr\{E_2 \cap E_1\}$. Usando a equação (C.14), temos

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\} .$$

A probabilidade $\Pr\{E_2 \mid E_1\}$ é igual a $1/(n - i + 1)$ porque, na linha 3, o algoritmo escolhe x_i aleatoriamente entre os $n - i + 1$ valores nas posições $A[i \mathrel{..} n]$. Desse modo, temos

$$\begin{aligned} \Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-1)!}{n!} . \end{aligned}$$

Término: No término, $i = n + 1$, e temos que o subarranjo $A[1 \mathrel{..} n]$ é uma n-permutação dada com probabilidade $(n - n)!/n! = 1/n!$.

Assim, RANDOMIZED-IN-PLACE produz uma permutação aleatória uniforme.

Muitas vezes, um algoritmo aleatorizado é o modo mais simples e mais eficiente de resolver um problema. Usaremos algoritmos aleatorizados ocasionalmente em todo o livro.

Exercícios

5.3-1 O professor Marceau faz objeções ao invariante de laço usado na prova do Lema 5.5. Ele questiona se o invariante de laço é verdadeiro antes da primeira iteração. Seu raciocínio é que poderíamos, com a mesma facilidade, declarar que um subarranjo vazio não contém nenhuma 0-permutação. Portanto, a probabilidade de um subarranjo vazio conter uma 0-permutação deve ser 0, o que invalida o invariante de laço antes da primeira iteração. Reescreva o procedimento RANDOMIZE-IN-PLACE, de modo que seu invariante de laço associado se aplique a um subarranjo não vazio antes da primeira iteração, e modifique a prova do Lema 5.5 para o seu procedimento.

5.3-2 O professor Kelp decide escrever um procedimento que produzirá aleatoriamente qualquer permutação além da permutação identidade. Ele propõe o seguinte procedimento:)

```
PERMUTE-WITHOUT-IDENTITY(A)
1 n = A.comprimento
2 for i = 1 to n − 1
3         trocar A[i] por A[RANDOM(i + 1, n)]
```

Esse código faz o que professor Kelp pretende?

5.3-3 Suponha que, em vez de trocar o elemento $A[i]$ por um elemento aleatório do subarranjo $A[i\ ..\ n]$, nós o trocássemos por um elemento aleatório de qualquer lugar no arranjo:

```
PERMUTE-WITH-ALL(A)
1 n = A.comprimento
2 for i = 1 to n
3         trocar A[i] por A[RANDOM(1, n)]
```

Esse código produz uma permutação aleatória uniforme? Justifique sua resposta.

5.3-4 O professor Armstrong sugere o seguinte procedimento para gerar uma permutação aleatória uniforme:

```
PERMUTE-BY-CICLING(A)
1 n = A.comprimento
2 seja B[1 . . n] um novo arranjo
3 deslocamento = RANDOM(1, n)
4 for i = 1 to n
5         dest = i + deslocamento
6         if dest > n
7                 dest = dest − n
8         B[dest] = A[i]
9 return B
```

Mostre que cada elemento $A[i]$ tem uma probabilidade $1/n$ de terminar em qualquer posição particular em B. Então, mostre que o professor Armstrong está equivocado, demonstrando que a permutação resultante não é uniformemente aleatória.

5.3-5 ★

Prove que, no arranjo P do procedimento PERMUTE-BY-SORTING, a probabilidade de todos os elementos serem únicos é, no mínimo, $1 - 1/n$.

5.3-6 Explique como implementar o algoritmo PERMUTE-BY-SORTING para tratar o caso no qual duas ou mais prioridades são idênticas. Isto é, seu algoritmo deve produzir uma permutação aleatória uniforme, mesmo que duas ou mais prioridades sejam idênticas.

5.3-7 Suponha que queiramos criar uma ***amostra aleatória*** do conjunto $\{1, 2, 3, ..., n\}$, isto é, um subconjunto S de m elementos, onde $0 \leq m \leq n$, tal que cada subconjunto m tenha igual probabilidade de ser criado. Um

modo seria fazer $A[i] = i$ para $i = 1, 2, 3, \ldots, n$, chamar RANDOMIZE-IN-PLACE(A), e depois tomar só os primeiros m elementos do arranjo. Esse método faria n chamadas ao procedimento RANDOM. Se n for muito maior do que m, podemos criar uma amostra aleatória com um número menor de chamadas a RANDOM. Mostre que o seguinte procedimento recursivo retorna um subconjunto aleatório m de S $\{1, 2, 3, ..., n\}$, no qual cada subconjunto m é igualmente provável, enquanto faz somente m chamadas a RANDOM:

```
RANDOM-SAMPLE(m, n)
1 if m == 0
2       return Ø
3 else S = RANDOM-SAMPLE(m − 1, n − 1)
4       i = RANDOM(1, n)
5       if i ∈ S
6             S = S ∪ {n}
7       else S = S ∪ {i}
8       return S
```

5.4 ★ ANÁLISE PROBABILÍSTICA E USOS ADICIONAIS DE VARIÁVEIS ALEATÓRIAS INDICADORAS

Esta seção avançada ilustra um pouco mais a análise probabilística por meio de quatro exemplos. O primeiro determina a probabilidade de, em uma sala com k pessoas, duas delas compartilharem a mesma data de aniversário. O segundo exemplo examina o que acontece quando lançamos aleatoriamente bolas em caixas. O terceiro investiga "sequências" de caras consecutivas no lançamento de moedas. O exemplo final analisa uma variante do problema da contratação, na qual você tem de tomar decisões sem entrevistar realmente todos os candidatos.

5.4.1 O PARADOXO DO ANIVERSÁRIO

Nosso primeiro exemplo é o ***paradoxo do aniversário***. Quantas pessoas devem estar em uma sala antes de existir uma chance de 50% de duas delas terem nascido no mesmo dia do ano? A resposta é um número de pessoas surpreendentemente pequeno. O paradoxo é que esse número é de fato muito menor que o número de dias do ano ou até menor que metade do número de dias do ano, como veremos.

Para responder à pergunta, indexamos as pessoas na sala com os inteiros 1, 2, ..., k, onde k é o número de pessoas na sala. Ignoramos a questão dos anos bissextos e supomos que todos os anos têm $n = 365$ dias. Para $i = 1, 2, ..., k$, seja b_i o dia do ano no qual cai o aniversário da pessoa i, onde $1 \leq b_i \leq n$. Supomos também que os aniversários estão uniformemente distribuídos pelos n dias do ano, de modo que $\Pr\{b_i = r\} = 1/n$ para $i = 1, 2, ..., k$ e $r = 1, 2, ..., n$.

A probabilidade de que duas pessoas dadas, digamos i e j, tenham datas de aniversário coincidentes depende do fato de a seleção aleatória de aniversários ser independente. De agora em diante, supomos que os aniversários são independentes, de modo que a probabilidade de o aniversário de i e o aniversário de j caírem ambos no dia r é

$$\begin{aligned} \Pr\{b_i = r \text{ e } b_j = r\} &= \Pr\{b_i = r\} \Pr\{b_j = r\} \\ &= 1/n^2 . \end{aligned}$$

Assim, a probabilidade de ambos caírem mesmo dia é

$$
\begin{aligned}
\Pr\{b_i = b_j\} &= \sum_{r=1}^{n} \Pr\{b_i = r \text{ e } b_j = r\} \\
&= \sum_{r=1}^{n} (1/n^2) \\
&= 1/n. \qquad (5.6)
\end{aligned}
$$

Mais intuitivamente, uma vez escolhido b_i, a probabilidade de b_j ser escolhido como o mesmo dia é $1/n$. Assim, a probabilidade de i e j terem o mesmo dia de aniversário é igual à probabilidade de o aniversário de um deles cair em um determinado dia. Porém, observe que essa coincidência depende de supor que os dias de aniversário são independentes.

Podemos analisar a probabilidade de, no mínimo, 2 entre k pessoas terem aniversários coincidentes examinando o evento complementar. A probabilidade de, no mínimo, dois dos aniversários coincidirem é 1 menos a probabilidade de todos os aniversários serem diferentes. O evento no qual k pessoas têm aniversários distintos é

$$B_k = \bigcap_{i=1}^{k} A_i,$$

onde A_i é o evento de o aniversário da pessoa i ser diferente do aniversário da pessoa j para todo $j < i$. Visto que podemos escrever $B_k = A_k \cap B_{k-1}$, obtemos da equação (C.16) a recorrência

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\}, \qquad (5.7)$$

onde tomamos $\Pr\{B_1\} = \Pr\{A_1\} = 1$ como uma condição inicial. Em outras palavras, a probabilidade de que $b_1, b_2, ..., b_k$ sejam aniversários distintos é a probabilidade de $b_1, b_2, ..., b_{k-1}$ serem aniversários distintos vezes a probabilidade de que $b_k \neq b_i$ para $i = 1, 2, ..., k - 1$, dado que $b_1, b_2, ..., b_{k-1}$ são distintos.

Se $b_1, b_2, ..., b_{k-1}$ são distintos, a probabilidade condicional de que $b_k \neq b_i$ para $i = 1, 2, ..., k - 1$ é $\Pr\{A_k \mid B_{k-1}\} = (n - k + 1)/n$, já que, dos n dias, há $n - (k - 1)$ que não são tomados. Aplicamos iterativamente a recorrência (5.7) para obter

$$
\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} \\
&= \Pr\{B_{k-2}\} \Pr\{A_{k-1} \mid B_{k-2}\} \Pr\{A_k \mid B_{k-1}\} \\
&\vdots \\
&= \Pr\{B_1\} \Pr\{A_2\} \mid B_1\} \Pr\{A_3 \mid B_2\} \ldots \Pr\{A_k \mid B_{k-1}\} \\
&= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right) \cdots \left(\frac{n-k+1}{n}\right) \\
&= 1 \cdot \left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right).
\end{aligned}
$$

A desigualdade (3.12), $1 + x \leq e^x$, nos dá

$$
\begin{aligned}
\Pr\{B_k\} &\leq e^{-1/n} e^{-2/n} \cdots e^{-(k-1)/n} \\
&= e^{-\sum_{i=1}^{k-1} i/n} \\
&= e^{-k(k-1)/2n} \\
&\leq 1/2
\end{aligned}
$$

quando $-k(k - 1)/2n \leq \ln(1/2)$. A probabilidade de que todos os k aniversários sejam distintos é, no máximo, 1/2 quando $k(k - 1) \geq 2n \ln 2$ ou, resolvendo a equação quadrática, quando $k \geq (1+ \sqrt{1+(8 \ln 2)}\, n/2$. Para $n = 365$,

devemos ter $k \geq 23$. Assim, se no mínimo 23 pessoas estiverem em uma sala, a probabilidade de que ao menos duas pessoas tenham a mesma data de aniversário é, no mínimo, 1/2. Em Marte, um ano dura 669 dias marcianos; então, seriam necessários 31 marcianos para conseguirmos o mesmo efeito.

Uma análise usando variáveis aleatórias indicadoras

Podemos usar variáveis aleatórias indicadoras para fornecer uma análise mais simples, embora aproximada, do paradoxo do aniversário. Para cada par (i, j) das k pessoas na sala, definimos a variável aleatória indicadora X_{ij}, para $1 \leq i < j \leq k$, por

$$
\begin{aligned}
X_{ij} &= I\{\text{a pessoa } i \text{ e a pessoa } j \text{ têm o mesmo dia de aniversário}\} \\
&= \begin{cases} 1 & \text{se a pessoa } i \text{ e a pessoa } j \text{ têm o mesmo dia de aniversário,} \\ 0 & \text{outro modo.} \end{cases}
\end{aligned}
$$

Pela equação (5.6), a probabilidade de duas pessoas terem aniversários coincidentes é $1/n$ e, assim, pelo Lema 5.1, temos

$$
\begin{aligned}
\mathrm{E}[X_{ij}] &= \Pr\{\text{pessoa } i \text{ e pessoa } j \text{ têm o mesmo dia aniversário}\} \\
&= 1/n\,.
\end{aligned}
$$

Sendo X a variável aleatória que conta o número de pares de indivíduos que têm a mesma data de aniversário, temos

$$
X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\,.
$$

Tomando as esperanças de ambos os lados e aplicando a linearidade de expectativa, obtemos

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right] \\
&= \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathrm{E}\left[X_{ij}\right] \\
&= \binom{k}{2} \frac{1}{n} \\
&= \frac{k(k-1)}{2n}\,.
\end{aligned}
$$

Portanto, quando $k(k - 1) \geq 2n$, o número esperado de pares de pessoas com a mesma data de aniversário é, no mínimo, 1. Assim, se tivermos no mínimo $\sqrt{2n} + 1$ indivíduos em uma sala, poderemos esperar que no mínimo dois deles façam aniversário no mesmo dia. Para $n = 365$, se $k = 28$, o número esperado de pares com o mesmo dia de aniversário é $(28 \cdot 27)/(2 \cdot 365) \approx 1{,}0356$. Assim, com no mínimo 28 pessoas, esperamos encontrar no mínimo um par de aniversários coincidentes. Em Marte, onde um ano corresponde a 669 dias marcianos, precisaríamos de no mínimo 38 marcianos.

A primeira análise, que usou somente probabilidades, determinou o número de pessoas necessárias para que a probabilidade de existir um par de datas de aniversário coincidentes exceda 1/2, e a segunda análise, que empregou

variáveis aleatórias indicadoras, determinou o número tal que o número esperado de aniversários coincidentes é 1. Embora os números exatos de pessoas sejam diferentes nas duas situações, eles são assintoticamente iguais: $\Theta(\sqrt{n})$

5.4.2 BOLAS E CAIXAS

Considere um processo no qual lançamos aleatoriamente bolas idênticas em b caixas, numeradas 1, 2, ..., b. Os lançamentos são independentes, e em cada lançamento a bola tem igual probabilidade de terminar em qualquer caixa. A probabilidade de uma bola lançada cair em qualquer caixa dada é $1/b$. Assim, o processo de lançamento de bolas é uma sequência de tentativas de Bernoulli (consulte o Apêndice C, Seção C.4) com uma probabilidade de sucesso $1/b$, onde sucesso significa que a bola cai na caixa dada. Esse modelo é particularmente útil para analisar hashing (espalhamento) (veja o Capítulo 11), e podemos responder a uma variedade de perguntas interessantes sobre o processo de lançamento de bolas. (O Problema C-1 faz perguntas adicionais sobre bolas e caixas.)

Quantas bolas caem em uma determinada caixa? O número de bolas que caem em uma caixa dada segue a distribuição binomial $b(k; n, 1/b)$. Se lançarmos n bolas, a equação (C.37) nos informa que o número esperado de bolas que caem na caixa dada é n/b.

Quantas bolas devemos lançar, em média, até que uma caixa dada contenha uma bola? O número de lançamentos até a caixa dada receber uma bola segue a distribuição geométrica com probabilidade $1/b$ e, pela equação (C.32), o número esperado de lançamentos até o sucesso é $1/(1/b) = b$.

Quantas bolas devemos lançar até toda caixa conter no mínimo uma bola? Vamos chamar um lançamento no qual uma bola cai em uma caixa vazia de "acerto". Queremos saber o número esperado n de lançamentos necessários para conseguir b acertos.

Usando os acertos, podemos dividir os n lançamentos em fases. A i-ésima fase consiste nos lançamentos depois do $(i - 1)$-ésimo acerto até o i-ésimo acerto. A primeira fase consiste no primeiro lançamento, já que um acerto já está garantido quando todas as caixas estão vazias. Para cada lançamento durante a i-ésima fase, $i - 1$ caixas contêm bolas e $b - i + 1$ caixas estão vazias. Assim, para cada lançamento na i-ésima fase, a probabilidade de obter um acerto é $(b - i + 1)/b$.

Seja n_i o número de lançamentos na i-ésima fase. Assim, o número de lançamentos exigidos para obter b acertos é $n = n \sum_{i=1}^{b} n_i$. Cada variável aleatória n_i tem uma distribuição geométrica com probabilidade de sucesso $(b - i + 1)/b$ e, pela equação (C.32), temos

$$\mathrm{E}[n_i] = \frac{b}{b - i + 1}.$$

Por linearidade de esperança,

$$
\begin{aligned}
\mathrm{E}[n] &= \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] \\
&= \sum_{i=1}^{b} \mathrm{E}[n_i] \\
&= \sum_{i=1}^{b} \frac{b}{b-i+1} \\
&= b\sum_{i=1}^{b} \frac{1}{i} \\
&= b(\ln b + O(1)) \quad \text{(pela equação (A.7))}.
\end{aligned}
$$

Portanto, são necessários aproximadamente $b \ln b$ lançamentos antes de podermos esperar que toda caixa tenha uma bola. Esse problema também é conhecido como ***problema do colecionador de cupons***, que diz que uma pessoa que tenta colecionar cada um de b cupons diferentes espera adquirir aproximadamente $b \ln b$ cupons obtidos aleatoriamente para ter sucesso.

5.4.3 SEQUÊNCIAS

Suponha que você lance uma moeda não viciada n vezes. Qual é a sequência mais longa de caras consecutivas que você espera ver? A resposta é Q(lg n), como mostra a análise a seguir.

Primeiro, provamos que o comprimento esperado da sequência mais longa de caras é $O(\lg n)$. A probabilidade de cada lançamento de moeda ser uma cara é 1/2. Seja A_{ik} o evento no qual uma sequência de caras de comprimento no mínimo k começa com o i-ésimo lançamento de moeda ou, mais precisamente, o evento no qual os k lançamentos consecutivos de moedas $i, i+1, ..., i+k-1$ produzem somente caras, onde $1 \le k \le n$ e $1 \le i \le n-k+1$. Como os lançamentos de moedas são mutuamente independentes, para qualquer evento dado A_{ik}, a probabilidade de todos os k lançamentos serem caras é

$$
\Pr\{A_{ik}\} = 1/2^k . \tag{5.8}
$$

e, portanto, a probabilidade de uma sequência de caras de comprimento no mínimo igual a 2 lg n começar na posição i é bastante pequena. Há, no máximo, $n - 2 \lg n + 1$ posições onde tal sequência pode começar. Portanto, a probabilidade de uma sequência de caras de comprimento no mínimo 2 lg n começar em qualquer lugar é

$$
\begin{aligned}
\Pr\left\{\bigcup_{i=1}^{n-2\lceil\lg n\rceil+1} A_{i,2\lceil\lg n\rceil}\right\} &\le \sum_{i=1}^{n-2\lceil\lg n\rceil+1} 1/n^2 \\
&< \sum_{i=1}^{n} 1/n^2 \\
&= 1/n,
\end{aligned} \tag{5.9}
$$

já que, pela desigualdade de Boole (C.19), a probabilidade de uma união de eventos é no máximo a soma das probabilidades dos eventos individuais. (Observe que a desigualdade de Boole é válida até mesmo para eventos como esses, que não são independentes.)

Agora, usamos a desigualdade (5.9) para limitar o comprimento da sequência mais longa. Para $j = 0, 1, 2, ..., n$, seja L_j o evento no qual a sequência mais longa de caras tem comprimento exatamente j, e seja L o comprimento da sequência mais longa. Pela definição de valor esperado, temos

$$E[L] = \sum_{j=0}^{n} j \Pr\{L_j\}. \tag{5.10}$$

Poderíamos tentar avaliar essa soma usando limites superiores para cada Pr $\{L_j\}$ semelhantes aos que foram calculados na desigualdade (5.9). Infelizmente, esse método produziria limites fracos. Porém, podemos usar certa intuição adquirida na análise anterior para obter um bom limite. Informalmente, observamos que não há nenhum termo individual no somatório da equação (5.10) para o qual ambos os fatores, j e Pr $\{L_j\}$, são grandes. Por quê? Quando $j \geq 2$ lg n, então Pr $\{L_j\}$ é muito pequeno e, quando $j < 2$ lg n, então j é razoavelmente pequeno. De um modo mais formal, notamos que os eventos L_j para $j = 0, 1, ..., n$ são disjuntos e, assim, a probabilidade de uma sequência de caras de comprimento no mínimo 2 lg n começar em qualquer lugar é $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\}$. Pela desigualdade (5.9), temos $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} < 1/n$. Além disso, observando que $\sum_{j=0}^{n} \Pr\{L_j\} = 1$, temos que $\sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} \leq 1$. Assim, obtemos

$$\begin{aligned}
E[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{2\lceil \lg n \rceil - 1} j \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} j \Pr\{L_j\} \\
&< \sum_{j=0}^{2\lceil \lg n \rceil - 1} (2\lceil \ln n \rceil) \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} n \Pr\{L_j\} \\
&= 2\lceil \ln n \rceil \sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} + n \sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} \\
&< 2\lceil \ln n \rceil \cdot 1 + n \cdot (1/n) \\
&= O(\lg n).
\end{aligned}$$

A probabilidade de uma sequência de caras exceder r lg n lançamentos diminui rapidamente com r. Para $r \geq 1$, a probabilidade de uma sequência de no mínimo r lg n caras começar na posição i é

$$\begin{aligned}
\Pr\{A_{i,r\lceil lg\, n \rceil}\} &= 1/2^{r\lceil lg\, n \rceil} \\
&\leq 1/n^{r}.
\end{aligned}$$

Assim, a probabilidade de a sequência mais longa ser no mínimo r lg n é no máximo igual a $n/n_{\mathrm{r}} = 1/n_{\mathrm{r}} - 1$ ou, o que é equivalente, a probabilidade de a sequência mais longa ter comprimento menor que r lg n é no mínimo $1 - 1/n_{\mathrm{r}} - 1$.

Como exemplo, para $n = 1.000$ lançamentos de moeda, a probabilidade de termos uma sequência de no mínimo 2 lg $n = 20$ caras é no máximo 1/ $n = 1/1.000$. A chance de termos uma sequência mais longa que 3 lg $n = 30$ caras é no máximo $1/n_2 = 1/1.000.000$.

Agora, vamos provar um limite complementar inferior: o comprimento esperado da sequência mais longa de caras em n lançamentos de moeda é (lg n). Para provar esse limite, procuramos sequências de comprimento s repartindo os n lançamentos em aproximadamente n/s grupos de s lançamentos cada. Se escolhermos $s =$ (lg n)/2, poderemos mostrar que é provável que no mínimo um desses grupos dê somente caras e, consequentemente, é provável que a sequência mais longa tenha comprimento no mínimo igual a $s =$ (lg n). Então, mostramos que a sequência mais longa tem comprimento esperado (lg n).

Repartimos os n lançamentos de moedas em no mínimo $n/(\lg n)/2$ grupos de $(\lg n)/2$ lançamentos consecutivos e limitamos a probabilidade de nenhum grupo dar somente caras. Pela equação (5.8), a probabilidade de o grupo que começa na posição i dar somente caras é

$$\begin{aligned} \Pr\{A_{i,\lfloor(\lg n)/2\rfloor}\} &= 1/2^{\lfloor(\lg n)/2\rfloor} \\ &\geq 1/\sqrt{n}. \end{aligned}$$

Então, a probabilidade de uma sequência de caras de comprimento no mínimo igual a $(\lg n)/2$ não começar na posição i é no máximo $1 - 1\sqrt{n}/$. Visto que os $n/(\lg n)/2$ grupos são formados por lançamentos de moedas mutuamente exclusivos e independentes, a probabilidade de cada um desses grupos *não ser* uma sequência de comprimento $(\lg n)/2$ é no máximo

$$\begin{aligned} (1-1/\sqrt{n})^{\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor} &\leq (1-1/\sqrt{n})^{n/\lfloor(\lg n)/2\rfloor-1} \\ &\leq (1-1/\sqrt{n})^{2n/\lg n-1} \\ &\leq e^{-(2n/\lg n-1)/\sqrt{n}} \\ &= O(e^{-\lg n}) \\ &= O(1/n). \end{aligned}$$

Para esse argumento, usamos a desigualdade (3.12), $1 + x \leq e_x$ e o fato, que seria bem interessante você verificar, de $(2n/\lg n - 1)/\sqrt{n} \geq \lg n$ para n suficientemente grande.

Assim, a probabilidade de a sequência mais longa exceder $(\lg n)/2$ é

$$\sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \Pr\{L_j\} \geq 1 - O(1/n). \tag{5.11}$$

Agora podemos calcular um limite inferior para o comprimento esperado da sequência mais longa, começando com a equação (5.10) e prosseguindo de modo semelhante à nossa análise do limite superior:

$$\begin{aligned} \mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\ &= \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} j \Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} j \Pr\{L_j\} \\ &\geq \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} 0 \cdot \Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \lfloor(\lg n)/2\rfloor \Pr\{L_j\} \\ &= 0 \cdot \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} \Pr\{L_j\} + \lfloor(\lg n)/2\rfloor \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \Pr\{L_j\} \\ &\geq 0 + \lfloor(\lg n)/2\rfloor(1-O(1/n)) \qquad \text{(pela desigualdade 5.11)} \\ &= (\ln n). \end{aligned}$$

Como ocorre no caso do paradoxo do aniversário, podemos obter uma análise mais simples, porém aproximada, usando variáveis aleatórias indicadoras. Seja $X_{ik} = \mathrm{I}\{A_{ik}\}$ a variável aleatória indicadora associada a uma sequência de caras de comprimento no mínimo k que começa com o i-ésimo lançamento da moeda. Para contar o número total de tais sequências, definimos

$$X = \sum_{i=1}^{n-k+1} X_{ik}.$$

Tomando esperanças e usando linearidade de esperança, temos

$$
\begin{aligned}
E[X] &= \left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\
&= \sum_{i=1}^{n-k+1} E[X_{ik}] \\
&= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\
&= \sum_{i=1}^{n-k+1} 1/2^k \\
&= \frac{n-k+1}{2^k}.
\end{aligned}
$$

Inserindo diversos valores para k, podemos calcular o número esperado de sequências de comprimento k. Se esse número for grande (muito maior que 1), esperamos que ocorram muitas sequências de comprimento k, e a probabilidade de ocorrer uma é alta. Se esse número for pequeno (muito menor que 1), esperamos que ocorra um pequeno número de sequências de comprimento k, e a probabilidade de ocorrer uma é baixa. Se $k = c \lg n$, para alguma constante positiva c, obtemos

$$
\begin{aligned}
E[X] &= \frac{n - c\lg n + 1}{2^{c \lg n}} \\
&= \frac{n - c\lg n + 1}{n^c} \\
&= \frac{1}{n^{c-1}} - \frac{(c \lg n - 1)/n}{n^c} \\
&= \Theta(1/n^{c-1}).
\end{aligned}
$$

Se c for grande, o número esperado de sequências de comprimento $c \lg n$ é pequeno, e concluímos que é improvável que elas ocorram. Por outro lado, se $c = 1/2$, então obtemos $E[X] = Q(1/n_{1/2} - 1) = Q(n_{1/2})$, e esperamos que exista um número grande de sequências de comprimento $(1/2) \lg n$. Portanto, é provável que ocorra uma sequência de tal comprimento. Só por essas estimativas grosseiras, já podemos concluir que o comprimento esperado da sequência mais longa é $Q(\lg n)$.

5.4.4 O PROBLEMA DA CONTRATAÇÃO ON-LINE

Como exemplo final, examinaremos uma variante do problema da contratação. Suponha agora que não desejamos entrevistar todos os candidatos para encontrar o melhor. Também não queremos contratar e demitir à medida que encontrarmos candidatos cada vez melhores. Em vez disso, estamos dispostos a aceitar um candidato próximo do

melhor em troca de contratar exatamente uma vez. Devemos obedecer a um requisito da empresa: após cada entrevista temos de oferecer imediatamente o cargo ao candidato ou rejeitá-lo também imediatamente. Qual é a permuta entre minimizar a quantidade de entrevistas e maximizar a qualidade do candidato contratado?

Podemos modelar esse problema da maneira ilustrada a seguir. Após a reunião com um candidato, podemos atribuir a cada um deles uma pontuação; seja *pontuação*(i) a pontuação dada ao i-ésimo candidato, e suponha que não há dois candidatos que recebam a mesma pontuação. Depois de entrevistar j candidatos, sabemos qual dos j candidatos tem a pontuação mais alta, mas não sabemos se algum dos $n - j$ candidatos restantes receberá uma pontuação mais alta do que aquele. Decidimos adotar a estratégia de selecionar um inteiro positivo $k < n$, entrevistar e depois rejeitar os primeiros k candidatos e, daí em diante, contratar o primeiro candidato que obtiver uma pontuação mais alta que todos os candidatos anteriores. Se notarmos que o candidato mais bem qualificado se encontrava entre os k primeiros entrevistados, contrataremos o n-ésimo candidato. Formalizamos essa estratégia no procedimento ON-LINE-MAXIMUM(k, n), que retorna o índice do candidato que desejamos contratar.

```
ON-LINE-MAXIMUM(k, n)
1    melhorpontuação = −∞
2    for i = 1 to k
3        if pontuação(i) > melhorpontuação
4            melhorpontuação = pontuação(i)
5    for i = k + 1 to n
6        if pontuação(i) > melhorpontuação
7            return i
8    return n
```

Desejamos determinar, para cada valor possível de k, a probabilidade de contratarmos o candidato mais bem qualificado. Então, escolhemos o k melhor possível e implementamos a estratégia com esse valor. Por enquanto, considere k fixo. Seja $M(j) = \max_{1 \le i \le j} \{pontuação(i)\}$ a pontuação máxima entre os candidatos 1 a j. Seja S o evento no qual temos sucesso na escolha do candidato mais bem qualificado, e seja S_i o evento no qual temos sucesso quando o candidato mais bem qualificado for o i-ésimo entrevistado. Visto que diversos S_i são disjuntos, temos que $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. Observando que nunca temos sucesso quando o candidato mais bem qualificado é um dos primeiros k, temos que $\Pr\{S_i\} = 0$ para $i = 1, 2, ..., k$. Assim, obtemos

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\}. \tag{5.12}$$

Agora calculamos $\Pr\{S_i\}$. Para ter sucesso quando o candidato mais bem qualificado é o i-ésimo, duas coisas devem acontecer. Primeiro, o candidato mais bem qualificado deve estar na posição i, um evento que denotamos por B_i. Segundo, o algoritmo não deve selecionar nenhum dos candidatos nas posições $k + 1$ a $i - 1$, o que acontece somente se, para cada j tal que $k + 1 \le j \le i - 1$, encontramos *pontuação*(j) < *melhorpontuação* na linha 6. (Como as pontuações são exclusivas, podemos ignorar a possibilidade de *pontuação*(j) = *melhorpontuação*.) Em outras palavras, todos os valores *pontuação*($k + 1$) até *pontuação*($i - 1$) devem ser menores que $M(k)$; se qualquer deles for maior que $M(k)$, em vez disso retornaremos o índice do primeiro que for maior. Usamos O_i para denotar o evento no qual nenhum dos candidatos nas posições $k + 1$ a $i - 1$ é escolhido. Felizmente, os dois eventos B_i e O_i são independentes. O evento O_i depende apenas da ordenação relativa dos valores nas posições 1 a $i - 1$, enquanto B_i depende apenas de o valor na posição i ser maior que todos os valores em todas as outras posições. A ordenação dos valores nas posições 1 a $i - 1$ não afeta o fato de o valor na posição i ser maior que todos eles, e o valor na posição i não afeta a ordenação dos valores nas posições 1 a $i - 1$. Assim, podemos aplicar a equação (C.15) para obter

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\} \Pr\{O_i\} .$$

A probabilidade Pr $\{B_i\}$ é claramente $1/n$, já que o máximo tem igual probabilidade de estar em qualquer uma das n posições. Para o evento O_i ocorrer, o valor máximo nas posições 1 a $i - 1$, que tem a mesma probabilidade de estar em qualquer uma dessas $i - 1$ posições, deve estar em uma das k primeiras posições. Consequentemente, Pr $\{O_i\} = k/(i - 1)$ e Pr $\{S_i\} = k/(n\,(i - 1))$. Usando a equação (5.12), temos

$$\begin{aligned}
\Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\
&= \sum_{i=k+1}^{n} \frac{k}{n(i-1)} \\
&= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} \\
&= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i}.
\end{aligned}$$

Aproximamos por integrais para limitar esse somatório acima e abaixo. Pelas desigualdades (A.12), temos

$$\int_{k}^{n} \frac{1}{x}\,dx \le \sum_{i=k}^{n-1} \frac{1}{i} \le \int_{k-1}^{n-1} \frac{1}{x}\,dx.$$

A avaliação dessas integrais definidas nos dá os limites

$$\frac{k}{n}(\ln n - \ln k) \le \Pr\{S\} \le \frac{k}{n}(\ln(n-1) - \ln(k-1)),$$

que representam um limite bastante preciso para Pr $\{S\}$. Como desejamos maximizar nossa probabilidade de sucesso, vamos focalizar a escolha do valor de k que maximiza o limite inferior para Pr $\{S\}$. (Além do mais, a expressão do limite inferior é mais fácil de maximizar que a expressão do limite superior.) Diferenciando a expressão $(k/n)\,(\ln n - \ln k)$ com relação a k, obtemos

$$\frac{1}{n}(\ln n - \ln k - 1).$$

Igualando essa derivada a 0, vemos que maximizamos o limite inferior para a probabilidade quando $\ln k = \ln n - 1 = \ln(n/e)$ ou, o que é equivalente, quando $k = n/e$. Assim, se implementarmos nossa estratégia com $k = n/e$, teremos sucesso na contratação do nosso candidato mais bem qualificado com probabilidade no mínimo $1/e$.

Exercícios

5.4-1 Quantas pessoas devem estar em uma sala antes que a probabilidade de alguém ter a mesma data de aniversário que você seja no mínimo 1/2? Quantas pessoas devem estar presentes antes de a probabilidade de no mínimo duas pessoas fazerem aniversário no dia 4 de julho ser maior que 1/2?

5.4-2 Suponha que lançamos bolas em b caixas até que alguma caixa contenha duas bolas. Cada lançamento é independente e cada bola tem a mesma probabilidade de cair em qualquer caixa. Qual é o número esperado

de lançamentos de bolas?

5.4-3 ★

Para a análise do paradoxo do aniversário, é importante que os aniversários sejam mutuamente independentes ou a independência aos pares é suficiente? Justifique sua resposta.

5.4-4 ★

Quantas pessoas devem ser convidadas para uma festa para ser provável que haja *três* pessoas com a mesma data de aniversário?

5.4-5 ★

Qual é a probabilidade de uma cadeia k em um conjunto de tamanho n formar uma k-permutação? Qual é a relação dessa pergunta com o paradoxo do aniversário?

5.4-6 ★

Suponha que n bolas sejam lançadas em n caixas, onde cada lançamento é independente e a bola tem igual probabilidade de cair em qualquer caixa. Qual é o número esperado de caixas vazias? Qual é o número esperado de caixas com exatamente uma bola?

5.4-7 ★

Aprimore o limite inferior para o comprimento da sequência mostrando que, em n lançamentos de uma moeda não viciada, a probabilidade de não ocorrer nenhuma sequência mais longa que $\lg n - 2 \lg \lg n$ caras consecutivas é menor que $1/n$.

Problemas

5-1 Contagem probabilística

Com um contador de b bits, normalmente só podemos contar até $2^b - 1$. Com a ***contagem probabilística*** de R. Morris, podemos contar até um valor muito maior, à custa de alguma perda de precisão.

Interpretamos um contador com valor i como uma contagem n_i para $i = 0, 1, ..., 2^b - 1$, onde os n_i formam uma sequência crescente de valores não negativos. Supomos que o valor inicial do contador é 0, representando uma contagem de $n_0 = 0$. A operação IN-CREMENT funciona de maneira probabilística em um contador que contém o valor i. Se $i = 2^b - 1$, então a operação informa um erro de estouro (overflow). Caso contrário, a operação INCREMENT aumenta 1 no contador com probabilidade $1/(n_{i+1} - n_i)$ ou deixa o contador inalterado com probabilidade $1 - 1/(n_{i+1} - n_i)$.

Se selecionarmos $n_i = i$ para todo $i \geq 0$, então o contador é um contador comum. Surgem situações mais interessantes se selecionarmos, digamos, $n_i = 2^{i-1}$ para $i > 0$ ou $n_i = F_i$ (o i-ésimo número de Fibonacci — consulte a Seção 3.2).

Para este problema, suponha que, $n_{2^b - 1}$ é suficientemente grande, de modo que a probabilidade de um erro de estouro seja desprezível.

a. Mostre que o valor esperado representado pelo contador após a execução de n operações INCREMENT é exatamente n.

b. A análise da variância da contagem representada pelo contador depende da sequência de n_i. Vamos considerar um caso simples: $n_i = 100i$ para todo $i \geq 0$. Estime a variância no valor representado pelo registrador após a execução de n operações INCREMENT.

5-2 Busca em um arranjo não ordenado

Este problema examina três algoritmos para procurar um valor x em um arranjo não ordenado A que consiste em n elementos.

Considere a seguinte estratégia aleatória: escolha um índice aleatório i em A. Se $A[i] = x$, então terminamos; caso contrário, continuamos a busca escolhendo um novo índice aleatório em A. Continuamos a escolher índices aleatórios em A até encontrarmos um índice j tal que $A[j] = x$ ou até verificarmos todos os elementos de A. Observe que, toda vez escolhemos um índice no conjunto inteiro de índices, é possível que examinemos um dado elemento mais de uma vez.

a. Escreva pseudocódigo para um procedimento RANDOM-SEARCH para implementar a estratégia citada. Certifique-se de que o seu algoritmo termina quando todos os índices em A já tiverem sido escolhidos.

b. Suponha que exista exatamente um índice i tal que $A[i] = x$. Qual é o número esperado de índices em A que devemos escolher antes de encontrarmos x e RANDOM--SEARCH terminar?

c. Generalizando sua solução para a parte (b), suponha que existam $k \geq 1$ índices i tais que $A[i] = x$. Qual é o número esperado de índices em A que devemos escolher antes de encontrarmos x e RANDOM-SEARCH terminar? Sua resposta deve ser uma função de n e k.

d. Suponha que não exista nenhum índice i tal que $A[i] = x$. Qual é o número esperado de índices em A que devemos escolher antes de verificarmos todos os elementos de A e RANDOM-SEARCH terminar?

Agora, considere um algoritmo de busca linear determinística, que denominamos DETERMINISTIC-SEARCH. Especificamente, o algoritmo procura A para x em ordem, considerando $A[1]$, $A[2]$, $A[3]$, ..., $A[n]$ até encontrar $A[i] = x$ ou chegar ao fim do arranjo. Considere todas as permutações possíveis do arranjo de entrada igualmente prováveis.

e. Suponha que exista exatamente um índice i tal que $A[i] = x$. Qual é o tempo de execução do caso médio de DETERMINISTIC-SEARCH? Qual é o tempo de execução do pior caso de DETERMINISTIC-SEARCH?

f. Generalizando sua solução para parte (e), suponha que existam $k \geq 1$ índices i tais que $A[i] = x$. Qual é o tempo de execução do caso médio de DETERMINISTIC-SEARCH? Qual é o tempo de execução do pior caso de DETERMINISTIC-SEARCH? Sua resposta deve ser uma função de n e k.

g. Suponha que não exista nenhum índice i tal que $A[i] = x$. Qual é o tempo de execução do caso médio de DETERMINISTIC-SEARCH? Qual é o tempo de execução do pior caso de DETERMINISTIC-SEARCH?

Finalmente, considere um algoritmo aleatorizado SCRAMBLE-SEARCH que funciona primeiro permutando aleatoriamente o arranjo de entrada e depois executando a busca linear determinística dada anteriormente para o arranjo permutado resultante.

h. Sendo k o número de índices i tais que $A[i] = x$, dê os tempos de execução esperado e do pior caso de SCRAMBLE-SEARCH para os casos nos quais $k = 0$ e $k = 1$. Generalize sua solução para tratar o caso no qual $k \geq 1$.

j. Qual dos três algoritmos de busca você usaria? Explique sua resposta.

NOTAS DO CAPÍTULO

Bollobás [54] Hofri [174] e Spencer [321] contêm grande número de técnicas probabilísticas avançadas. As vantagens dos algoritmos aleatorizados são discutidas e pesquisadas por Karp [200] e Rabin [288]. O livro didático de Motwani e Raghavan [262] apresenta um tratamento extensivo de algoritmos aleatorizados.

Diversas variantes do problema da contratação têm sido amplamente estudadas. Esses problemas são mais comumente referidos como "problemas da secretária". Um exemplo de trabalho nessa área é o artigo de Ajtai, Meggido e Waarts [11].

Parte

ORDENAÇÃO E ESTATÍSTICAS DE ORDEM

INTRODUÇÃO

Esta parte apresenta vários algoritmos que resolvem o ***problema de ordenação*** a seguir:

Entrada: Uma sequência de n números $\langle a_1, a_2, ..., a_n \rangle$.

Saída: Uma permutação (reordenação) $(a'_1, a'_2, ..., a'_n)$ da sequência de entrada tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

A sequência de entrada normalmente é um arranjo de n elementos, embora possa ser representada de algum outro modo, como uma lista ligada.

A ESTRUTURA DOS DADOS

Na prática, os números que devem ser ordenados raramente são valores isolados. Em geral, cada um deles é parte de uma coleção de dados denominada ***registro***. Cada registro contém uma ***chave***, que é o valor a ser ordenado. O restante do registro consiste em ***dados satélites***, que normalmente são transportados junto com a **chave**. Na prática, quando um algoritmo de ordenação permuta as chaves, também deve permutar os dados satélites. Se cada registro incluir grande quantidade de dados satélites, muitas vezes permutamos um arranjo de ponteiros para os registros em vez dos próprios registros, para minimizar a movimentação de dados.

De certo modo, são esses detalhes de implementação que distinguem um algoritmo de um programa completamente desenvolvido. Um algoritmo de ordenação descreve o método pelo qual determinamos a sequência ordenada, independentemente de estarmos ordenando números individuais ou grandes registros contendo muitos bytes de dados satélites. Assim, quando focalizamos o problema de ordenação, em geral consideramos que a entrada consiste apenas em números. A tradução de um algoritmo para ordenação de números em um programa para ordenação de registros é conceitualmente direta, embora em uma situação específica de engenharia possam surgir outras sutilezas que fazem da tarefa real de programação um desafio.

Por que ordenar?

Muitos cientistas de computação consideram a ordenação o problema mais fundamental no estudo de algoritmos. Há várias razões:

- Às vezes, uma aplicação tem uma necessidade inerente de ordenar informações. Por exemplo, para preparar extratos de clientes, os bancos precisam ordenar os cheques pelo número do cheque.
- Os algoritmos frequentemente usam a ordenação como uma subrotina chave. Por exemplo, um programa que apresenta objetos gráficos dispostos em camadas uns sobre os outros talvez tenha de ordenar os objetos de

acordo com uma relação “acima” para poder desenhar esses objetos de baixo para cima. Neste texto veremos numerosos algoritmos que utilizam a ordenação como uma subrotina.

- Podemos escolher entre uma ampla variedade de algoritmos de ordenação, e eles empregam um rico conjunto de técnicas. De fato, muitas técnicas importantes usadas em projeto de algoritmos aparecem no corpo de algoritmos de ordenação que foram desenvolvidos ao longo dos anos. Assim, a ordenação também é um problema de interesse histórico.
- Podemos demonstrar um limite inferior não trivial para a ordenação (como faremos no Capítulo 8). Nossos melhores limites superiores correspondem ao limite inferior assintoticamente e, assim, sabemos que nossos algoritmos de ordenação são assintoticamente ótimos. Além disso, podemos usar o limite inferior de ordenação para demonstrar limites inferiores para alguns outros problemas.
- Muitas questões de engenharia vêm à tona na implementação de algoritmos de ordenação. O programa de ordenação mais rápido para determinada situação pode depender de muitos fatores, como o conhecimento anterior das chaves e dos dados satélites, da hierarquia de memória (caches e memória virtual) do computador hospedeiro e do ambiente de software. Muitas dessas questões são mais bem tratadas no nível algorítmico, em vez de “retocar” o código.

Algoritmos de ordenação

Apresentamos dois algoritmos para ordenação de n números reais no Capítulo 2. A ordenação por inserção leva o tempo Q(n_2) no pior caso. Porém, como seus laços internos são compactos, ela é um algoritmo rápido de ordenação no lugar para pequenos tamanhos de entrada. (Lembre-se de que um algoritmo de ordenação executa a ordenação ***no lugar*** se somente um número constante de elementos do arranjo de entrada estiver a cada vez armazenado fora do arranjo.)

A ordenação por intercalação tem um tempo de execução assintótico melhor, Q(n lg n), mas o procedimento MERGE que ela utiliza não funciona no lugar.

Nesta parte, apresentaremos mais dois algoritmos que ordenam números reais arbitrários. A ordenação por heap, apresentada no Capítulo 6, ordena n números no lugar, no tempo $O(n$ lg $n)$. Ela usa uma importante estrutura de dados, denominada heap, com a qual também podemos implementar uma fila de prioridades.

O quicksort, no Capítulo 7, também ordena n números no lugar, mas seu tempo de execução do pior caso é Q(n_2). Contudo, seu tempo de execução esperado é Q(n lg n), e em geral ele supera a ordenação por heap na prática. Como a ordenação por inserção, o quicksort tem um código compacto e, assim, o fator constante oculto em seu tempo de execução é pequeno. Ele é um algoritmo popular para ordenação de grandes arranjos de entrada.

Ordenação por inserção, ordenação por intercalação, ordenação por heap e quicksort são ordenações por comparação: determinam a sequência ordenada de um arranjo de entrada por comparação de elementos. O Capítulo 8 começa introduzindo o modelo de árvore de decisão para estudar as limitações de desempenho de ordenações por comparação. Usando esse modelo, provamos um limite inferior de W(n lg n) no tempo de execução do pior caso de qualquer ordenação por comparação para n entradas, mostrando assim que a ordenação por heap e a ordenação por intercalação são ordenações por comparação assintoticamente ótimas.

Em seguida, o Capítulo 8 mostra que poderemos superar esse limite inferior de W(n lg n) se for possível reunir informações sobre a sequência ordenada da entrada por outros meios além da comparação de elementos. Por exemplo, o algoritmo de ordenação por contagem considera que os números da entrada estão no conjunto $\{0, 1, \ldots, k\}$. Usando a indexação de arranjos como ferramenta para determinar a ordem relativa, a ordenação por contagem pode ordenar n números no tempo Q($k + n$). Assim, quando $k = O(n)$, a ordenação por contagem é executada em tempo linear no tamanho do arranjo de entrada. Um algoritmo relacionado, de ordenação digital (radix sort), pode ser usado para estender a faixa da ordenação por contagem. Se houver n inteiros para ordenar, cada inteiro tiver d dígitos e cada dígito puder adotar até k valores possíveis, a ordenação digital poderá ordenar os números no tempo Q($d(n + k)$). Quando d é uma constante e k é $O(n)$, a ordenação digital é executada em tempo linear. Um terceiro algoritmo, ordenação por balde (ordenação por balde) requer o conhecimento da distribuição probabilística dos números no arranjo de entrada.

Ele pode ordenar n números reais distribuídos uniformemente no intervalo meio aberto [0, 1) no tempo do caso médio $O(n)$.

A tabela a seguir resume os tempos de execução dos algoritmos de ordenação dos Capítulos 2 e 6–8. Como sempre, n denota o número de itens a ordenar. Para a ordenação por contagem, os itens a ordenar são inteiros no conjunto $\{0, 1, \ldots, k\}$. Para ordenação digital, cada item é um número com d dígitos, onde cada dígito adota k valores possíveis. Para a ordenação por balde, consideramos que as chaves são números reais uniformemente distribuídos no intervalo meio aberto [0, 1). A coluna da extrema direita dá o tempo de execução do caso médio ou esperado e indica a qual ela se refere quando for diferente do tempo de execução do pior caso. Omitimos o tempo de execução do caso médio da ordenação por heap porque não o analisaremos neste livro.

Algoritmo	Tempo de execução do pior caso	Tempo de execução do caso médio/esperado
Ordenação por inserção	$Q(n_2)$	$Q(n_2)$
Ordenação por intercalação	$Q(n \log n)$	$Q(n \log n)$
Ordenação por heap	$O(n \log n)$	—
Quicksort	$Q(n_2)$	$Q(n \log n)$ (esperado)
Ordenação por contagem	$Q(k + n)$	$Q(k + n)$
Ordenação digital	$Q(d(n + k))$	$Q(d(n + k))$
Ordenação por balde	$Q(n_2)$	$Q(n)$ (caso médio)

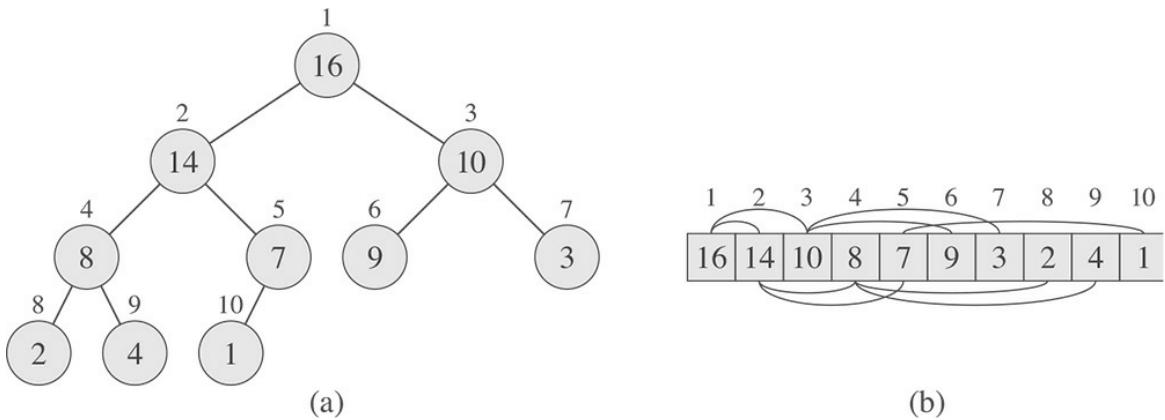
Estatísticas de ordem

A i-ésima estatística de ordem de um conjunto de n números é o i-ésimo menor número no conjunto. É claro que podemos selecionar a i-ésima estatística de ordem ordenando a entrada e indexando o i-ésimo elemento da saída. Sem nada supor sobre a distribuição da entrada, esse método é executado no tempo $W(n \lg n)$, como mostra o limite inferior demonstrado no Capítulo 8.

No Capítulo 9, mostramos que podemos determinar o i-ésimo menor elemento no tempo $O(n)$, mesmo quando os elementos são números reais arbitrários. Apresentamos um algoritmo aleatorizado com pseudocódigo compacto que é executado no tempo $Q(n_2)$ no pior caso, mas cujo tempo de execução esperado é $O(n)$. Também damos um algoritmo mais complicado que é executado no tempo $O(n)$ no pior caso.

Conhecimentos necessários

Embora a maioria das seções desta parte não dependa de conceitos matemáticos difíceis, algumas seções exigem certa sofisticação matemática. Em particular, as análises do quicksort, ordenação por balde e algoritmo de estatística de ordem utilizam probabilidade, que revisamos no Apêndice C; o material sobre análise probabilística e algoritmos aleatórios é estudado no Capítulo 5. A análise do algoritmo de tempo linear do pior caso para estatísticas de ordem envolve matemática um pouco mais sofisticada que as outras análises do pior caso apresentadas nesta parte.

6 ORDENAÇÃO POR HEAP

Neste capítulo, introduzimos outro algoritmo de ordenação: ordenação por heap. Como a ordenação por intercalação, mas diferente da ordenação por inserção, o tempo de execução da ordenação por heap é $O(n \lg n)$. Como a ordenação por inserção, mas diferentemente da ordenação por intercalação, a ordenação por heap ordena no lugar: apenas um número constante de elementos do arranjo é armazenado fora do arranjo de entrada em qualquer instante. Assim, a ordenação por heap combina os melhores atributos dos dois algoritmos de ordenação que já discutimos.

A ordenação por heap também introduz outra técnica de projeto de algoritmos: a utilização de uma estrutura de dados, nesse caso uma estrutura que denominamos "heap" para gerenciar informações. A estrutura de dados heap não é útil apenas para a ordenação por heap, ela também cria uma eficiente fila de prioridades. A estrutura de dados heap reaparecerá em algoritmos em capítulos posteriores.

O termo "heap" foi cunhado originalmente no contexto da ordenação por heap, mas desde então passou a se referir também a "armazenamento com coleta de lixo", tal como dado pelas linguagens de programação Lisp e Java. Nossa estrutura de dados heap *não* é armazenamento com coleta de lixo e, sempre que mencionarmos heaps neste livro, o termo significa a estrutura de dados definida neste capítulo.

6.1 HEAPS

A estrutura de dados ***heap*** (***binário***) é um objeto arranjo que pode ser visto como uma árvore binária quase completa (veja Seção B.5.3), como mostra a Figura 6.1. Cada nó da árvore corresponde a um elemento do arranjo. A árvore está completamente preenchida em todos os níveis, exceto possivelmente no nível mais baixo, que é preenchido a partir da esquerda até um ponto. Um arranjo A que representa um heap é um objeto com dois atributos: $A \cdot comprimento$, que (como sempre) dá o número de elementos no arranjo, e $A \cdot tamanho\text{-}do\text{-}heap$, que representa quantos elementos no heap estão armazenados dentro do arranjo A. Isto é, embora $A[1 \,..\, A \cdot comprimento]$ possa conter números, só os elementos em $A[A \cdot tamanho\text{-}do\text{-}heap]$, onde $A \cdot tamanho\text{-}do\text{-}heap \leq A \cdot comprimento$, são elementos válidos do heap. A raiz da árvore é $A[1]$ e, dado o índice i de um nó, podemos calcular facilmente os índices de seu pai, do filho à esquerda e do filho à direita:

Figura 6.1 Um heap de máximo visto como (**a**) uma árvore binária e (**b**) um arranjo. O número dentro do círculo em cada nó na árvore é o valor armazenado nesse nó. O número acima de um nó é o índice correspondente no arranjo. Acima e abaixo do arranjo há linhas que mostram relacionamentos pai-filho; os pais estão sempre à esquerda de seus filhos. A árvore tem altura três; o nó no índice 4 (com valor 8) tem altura um.

PARENT(i)
1 **return** $[i/2]$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

Na maioria dos computadores, o procedimento LEFT pode calcular $2i$ em uma única instrução, simplesmente deslocando a representação binária de i uma posição de bit para a esquerda. De modo semelhante, o procedimento RIGHT pode calcular rapidamente $2i + 1$ deslocando a representação binária de i uma posição de bit para a esquerda e depois inserindo 1 como o bit de ordem baixa. O procedimento PARENT pode calcular $i/2$ deslocando i uma posição de bit para a direita. Uma boa implementação de ordenação por heap frequentemente implementa esses procedimentos como “macros” ou “em linha”.

Existem dois tipos de heaps binários: heaps de máximo e heaps de mínimo. Em ambos os tipos, os valores nos nós satisfazem a uma ***propriedade de heap***, cujos detalhes específicos dependem do tipo de heap. Em um ***heap de máximo***, a ***propriedade de heap de máximo*** é que, para todo nó i exceto a raiz,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

isto é, o valor de um nó é, no máximo, o valor de seu pai. Assim, o maior elemento em um heap de máximo é armazenado na raiz, e a subárvore que tem raiz em um nó contém valores menores que o próprio nó. Um ***heap* de *mínimo*** é organizado de modo oposto; a ***propriedade de heap de mínimo*** é que, para todo nó i exceto a raiz,

$$A[\text{PARENT}(i)] \leq A[i] .$$

O menor elemento em um heap de mínimo está na raiz.

Para o algoritmo de ordenação por heap, usamos heaps de máximo. Heaps de mínimo são comumente empregados em filas de prioridades, que discutiremos na Seção 6.5. Seremos precisos ao especificar se necessitamos de um heap de máximo ou de um heap de mínimo para qualquer aplicação particular e, quando as propriedades se aplicarem tanto a heaps de máximo quanto a heaps de mínimo, simplesmente usaremos o termo “heap”.

Visualizando um heap como uma árvore, definimos a ***altura*** de um nó em um heap como o número de arestas no caminho descendente simples mais longo desde o nó até uma folha e definiremos a altura do heap como a altura de sua

raiz. Visto que um heap de n elementos é baseado em uma árvore binária completa, sua altura é Q(lg n) (veja Exercício 6.1-2). Veremos que as operações básicas em heaps são executadas em tempo que é, no máximo, proporcional à altura da árvore e, assim, demoram um tempo $O(\lg n)$. O restante deste capítulo apresenta alguns procedimentos básicos e mostra como eles são usados em um algoritmo de ordenação e uma estrutura de dados de fila de prioridades.

- O procedimento MAX-HEAPIFY, que roda no tempo $O(\lg n)$, é a chave para manter a propriedade de heap de máximo (6.1).
- O procedimento BUILD-MAX-HEAP, executado em tempo linear, produz um heap de máximo a partir de um arranjo de entrada não ordenado.
- O procedimento ordenação por heap, executado no tempo $O(n \lg n)$, ordena um arranjo no lugar.
- Os procedimentos MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY e HEAP--MAXIMUM, que rodam em tempo $O(\lg n)$, permitem que a estrutura de dados heap implemente uma fila de prioridades.

Exercícios

6.1-1 Quais são os números mínimo e máximo de elementos em um heap de altura h?

6.1-2 Mostre que um heap de n elementos tem altura lg n.

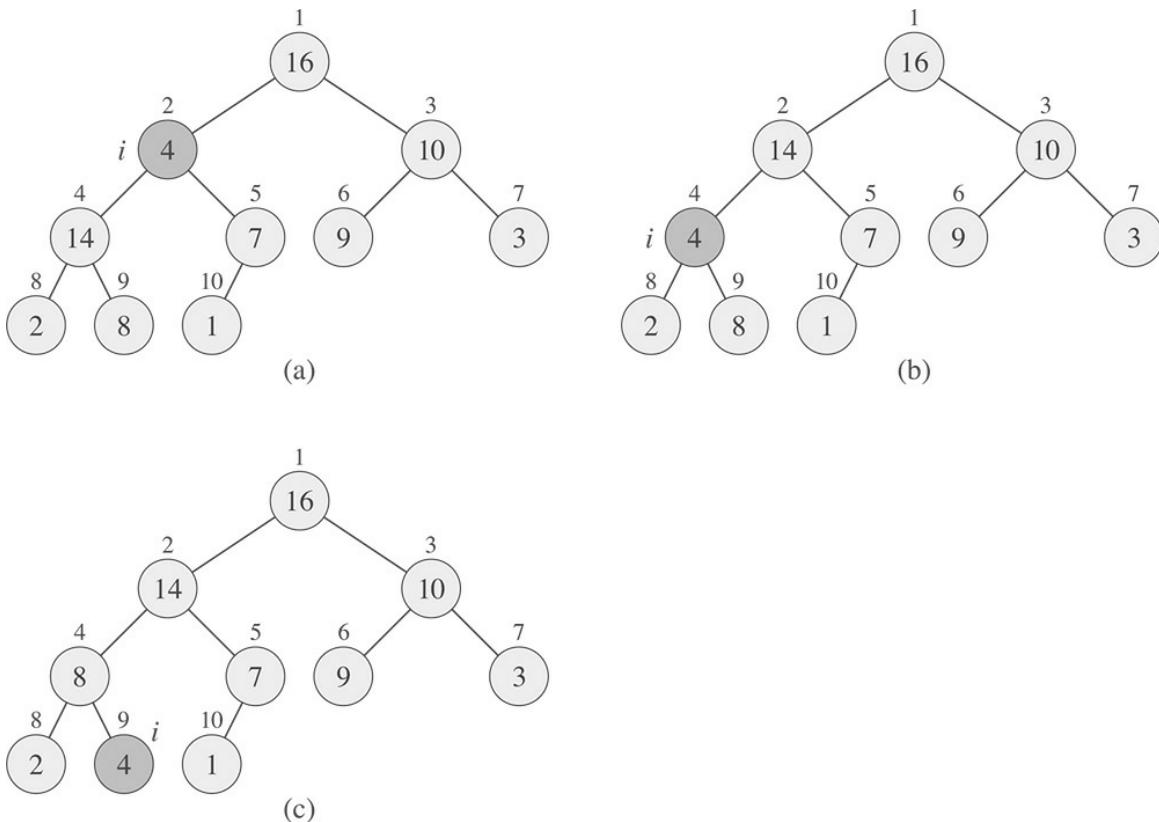
6.1-3 Mostre que, em qualquer subárvore de um heap de máximo, a raiz da subárvore contém o maior valor que ocorre em qualquer lugar nessa subárvore.

6.1-4 Em que lugar de heap de máximo o menor elemento poderia residir, considerando que todos os elementos sejam distintos?

6.1-5 Um arranjo que está em sequência ordenada é um heap de mínimo?

6.1-6 A sequência (23, 17, 14, 6, 13, 10, 1, 5, 7, 12) é um heap de máximo?

6.1-7 Mostre que, com a representação de arranjo para ordenar um heap de n elementos, as folhas são os nós indexados por $n/2 + 1$, $n/2 + 2$, ..., n.

6.2 MANUTENÇÃO DA PROPRIEDADE DE HEAP

Para manter a propriedade de heap de máximo, chamamos o procedimento MAX-HEAPIFY. Suas entradas são um arranjo A e um índice i para o arranjo. Quando chamado, MAX-HEAPIFY considera que as árvores binárias com raízes em LEFT(i) e RIGHT(i) são heaps de máximo, mas que $A[i]$ pode ser menor que seus filhos, o que viola a propriedade de heap de máximo. MAX-HEAPIFY permite que o valor em $A[i]$ "flutue para baixo" no heap de máximo, de modo que a subárvore com raiz no índice i obedeça à propriedade do heap de máximo.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A·tamanho-do-heap e A[l] > A[i]
 4       maior = l
 5  else maior = i
 6  if r ≤ A·tamanho-do-heap e A[r] > A[maior]
 7     maior = r
 8  if maior ≠ i
 9        trocar A[i] com A[maior]
10        MAX-HEAPIFY(A, maior)
```

Figura 6.2 Ação de MAX-HEAPIFY(A, 2), onde $A \cdot$ *tamanho-do-heap* $[A] = 10$. (**a**) Configuração inicial, com $A[2]$ no nó $i = 2$, violando a propriedade de heap de máximo, já que ele não é maior que os filhos. A propriedade de heap de máximo é restabelecida para o nó 2 em (**b**) pela troca de $A[2]$ por $A[4]$, o que destrói a propriedade de heap de máximo para o nó 4. A chamada recursiva MAX-HEAPIFY(A, 4) agora tem $i = 4$. Após trocar $A[4]$ por $A[9]$, como mostramos em (**c**), o nó 4 é corrigido, e a chamada recursiva a MAX-HEAPIFY(A, 9) não produz nenhuma mudança adicional na estrutura de dados.

A Figura 6.2 ilustra a ação de MAX-HEAPIFY. Em cada etapa, o maior dos elementos $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$ é determinado, e seu índice é armazenado em *maior*. Se $A[i]$ é maior, a subárvore com raiz no nó i já é um heap de máximo, e o procedimento termina. Caso contrário, um dos dois filhos tem o maior elemento, e $A[i]$ é trocado por $A[maior]$, fazendo com que o nó i e seus filhos satisfaçam a propriedade de heap de máximo. Porém, agora o nó

indexado por *maior* tem o valor original $A[i]$ e, assim, a subárvore com raiz em *maior* poderia violar a propriedade de heap de máximo. Em consequência disso, chamamos MAX-HEAPIFY recursivamente nessa subárvore.

O tempo de execução de MAX-HEAPIFY em uma subárvore de tamanho n com raiz em um dado nó i é o tempo Q(1) para corrigir as relações entre os elementos $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$, mais o tempo para executar MAX-HEAPIFY em uma subárvore com raiz em um dos filhos do nó i (considerando que a chamada recursiva ocorre). Cada uma das subárvores dos filhos tem, no máximo, tamanho igual a $2n/3$ — o pior caso ocorre quando a última linha da árvore está exatamente metade cheia — e, portanto, podemos descrever o tempo de execução de MAX-HEAPIFY pela recorrência

$$T(n) \leq T(2n/3) + \Theta(1) .$$

A solução para essa recorrência, de acordo com o caso 2 do teorema mestre (Teorema 4.1), é $T(n) = O(\lg n)$. Como alternativa, podemos caracterizar o tempo de execução de MAX-HEAPIFY em um nó de altura h como $O(h)$.

Exercícios

6.2-1 Usando a Figura 6.2 como modelo, ilustre a operação de MAX-HEAPIFY$(A, 3)$ sobre o arranjo $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2-2 Começando com o procedimento MAX-HEAPIFY, escreva pseudocódigo para o procedimento MIN-HEAPIFY(A, i), que executa a manipulação correspondente sobre um heap de mínimo. Compare o tempo de execução de MIN-HEAPIFY com o de MAX-HEAPIFY.

6.2-3 Qual é o efeito de chamar MAX-HEAPIFY(A, i) quando o elemento $A[i]$ é maior que seus filhos?

6.2-4 Qual é o efeito de chamar MAX-HEAPIFY(A, i) para $i > A \cdot \textit{tamanho-do-heap}/2$?

6.2-5 O código para MAX-HEAPIFY é bastante eficiente em termos de fatores constantes, exceto possivelmente para a chamada recursiva na linha 10, que poderia fazer com que alguns compiladores produzissem código ineficiente. Escreva um MAX-HEAPIFY eficiente que use um constructo de controle iterativo (um laço) em vez de recursão.

6.2-6 Mostre que o tempo de execução do pior caso de MAX-HEAPIFY para um heap de tamanho n é W(lg n). (*Sugestão*: Para um heap com n nós, dê valores de nós que façam MAX-HEAPIFY ser chamado recursivamente em todo nó em um caminho desde a raiz até uma folha.)

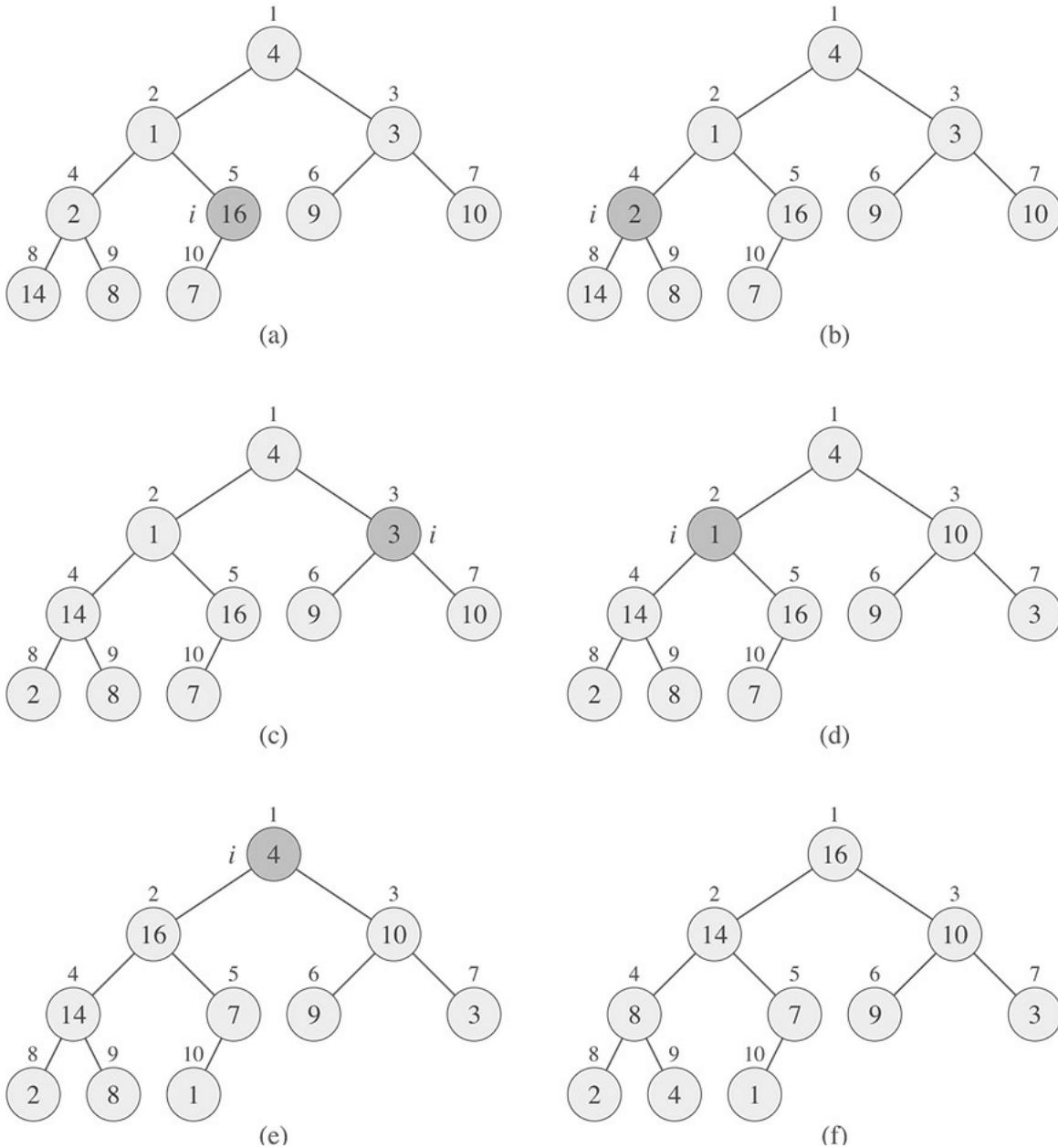
6.3 CONSTRUÇÃO DE UM HEAP

Podemos usar o procedimento MAX-HEAPIFY de baixo para cima para converter um arranjo $A[1 \mathinner{..} n]$, onde $n = A \cdot \textit{comprimento}$, em um heap de máximo. Pelo Exercício 6.1-7, os elementos no subarranjo $A[(\lfloor n/2 \rfloor + 1) \mathinner{..} n]$ são folhas da árvore e, portanto, já de início, cada um deles é um heap de 1 elemento. O procedimento BUILD-MAX-HEAP percorre os nós restantes da árvore e executa MAX-HEAPIFY sobre cada um.

```
BUILD-MAX-HEAP(A)
1  A·tamanho-do-heap = A·comprimento
2  for i = ⌊comprimento[A]/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

A Figura 6.3 mostra um exemplo da ação de BUILD-MAX-HEAP.

Para mostrar por que BUILD-MAX-HEAP funciona corretamente, usamos o seguinte invariante de laço:

No começo de cada iteração do laço **for** das linhas 2-3, cada nó $i + 1$, $i + 2$, …, n é a raiz de um heap de máximo.

Precisamos mostrar que esse invariante é verdadeiro antes da primeira iteração do laço, que cada iteração do laço mantém o invariante e que o invariante dá uma propriedade útil para mostrar a correção quando o laço termina.

Inicialização: Antes da primeira iteração do laço, $i = n/2$. Cada nó $n/2 + 1$, $n/2 + 2$, ..., n é uma folha, e é portanto a raiz de um heap de máximo trivial.

Manutenção: Para ver que cada iteração mantém o invariante de laço, observe que os filhos do nó i são numerados com valores mais altos que i. Assim, pelo invariante de laço, ambos são raízes de heaps de máximo. Essa é precisamente a condição exigida para a chamada MAX-HEAPIFY(A, i) para fazer do nó i uma raiz de heap de máximo. Além disso, a chamada a MAX-HEAPIFY preserva a propriedade de que os nós $i + 1$, $i + 2$, ..., n são raízes de heaps de máximo. Decrementar i na atualização do laço **for** restabelece o invariante de laço para a próxima iteração.

Figura 6.3 Operação de Build-Max-Heap, mostrando a estrutura de dados antes da chamada a Max-Heapify na linha 3 de Build-Max-Heap. (**a**) Um arranjo de entrada de 10 elementos A e a árvore binária que ele representa. A figura mostra que o índice de laço i se refere ao nó 5 antes da chamada Max-Heapify(A, i). (**b**) A estrutura de dados resultante. O índice de laço i para a próxima iteração aponta para o nó 4. (**c**)–(**e**) Iterações subsequentes do laço **for** em Build-Max-Heap. Observe que, sempre que Max-Heapify é chamado em um nó, as duas subárvores desse nó são heaps de máximo. (**f**) O heap de máximo após o término de Build-Max-Heap.

Término: No término, $i = 0$. Pelo invariante de laço, cada nó 1, 2, ..., n é a raiz de um heap de máximo. Em particular, o nó 1 é uma raiz.

Podemos calcular um limite superior simples para o tempo de execução de Build-Max-Heap da seguinte maneira: cada chamada a Max-Heapify custa o tempo $O(\lg n)$, e Build-max-Heap faz $O(n)$ dessas chamadas. Assim, o tempo de execução é $O(n \lg n)$. Esse limite superior, embora correto, não é assintoticamente restrito.

Podemos derivar um limite mais restrito observando que o tempo de execução de Max-Heapify em um nó varia com a altura do nó na árvore, e as alturas na maioria dos nós são pequenas. Nossa análise mais restrita se baseia nas

propriedades de que um heap de n elementos tem altura lg n (veja Exercício 6.1-2) e, no máximo $n/2^{h+1}$, nós de qualquer altura h (veja Exercício 6.3-3).

O tempo exigido por MAX-HEAPIFY quando chamado em um nó de altura h é $O(h)$; assim, podemos expressar o custo total de BUILD-MAX-HEAP limitado por cima por

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

Avaliamos o último somatório substituindo $x = 1/2$ na fórmula (A.8), o que produz

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$$

Assim, podemos limitar o tempo de execução de BUILD-MAX-HEAP como

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n).$$

Consequentemente, podemos construir um heap de máximo a partir de um arranjo não ordenado em tempo linear.

Podemos construir um heap de mínimo pelo procedimento BUILD-MIN-HEAP, que é igual a BUILD-MAX-HEAP, a não ser pela chamada a MAX-HEAPIFY na linha 3, que é substituída por uma chamada a MIN-HEAPIFY (veja Exercício 6.2-2). BUILD-MIN-HEAP produz um heap de mínimo a partir de um arranjo linear não ordenado em tempo linear.

Exercícios

6.3-1 Usando a Figura 6.3 como modelo, ilustre a operação de BUILD-MAX-HEAP no arranjo $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2 Por que queremos que o índice de laço i na linha 2 de BUILD-MAX-HEAP diminua de $A \cdot comprimento/2$ até 1, em vez de aumentar de 1 até $A \cdot comprimento/2$?

6.3-3 Mostre que existem, no máximo, $n/2^{h+1}$ nós de altura h em qualquer heap de n elementos.

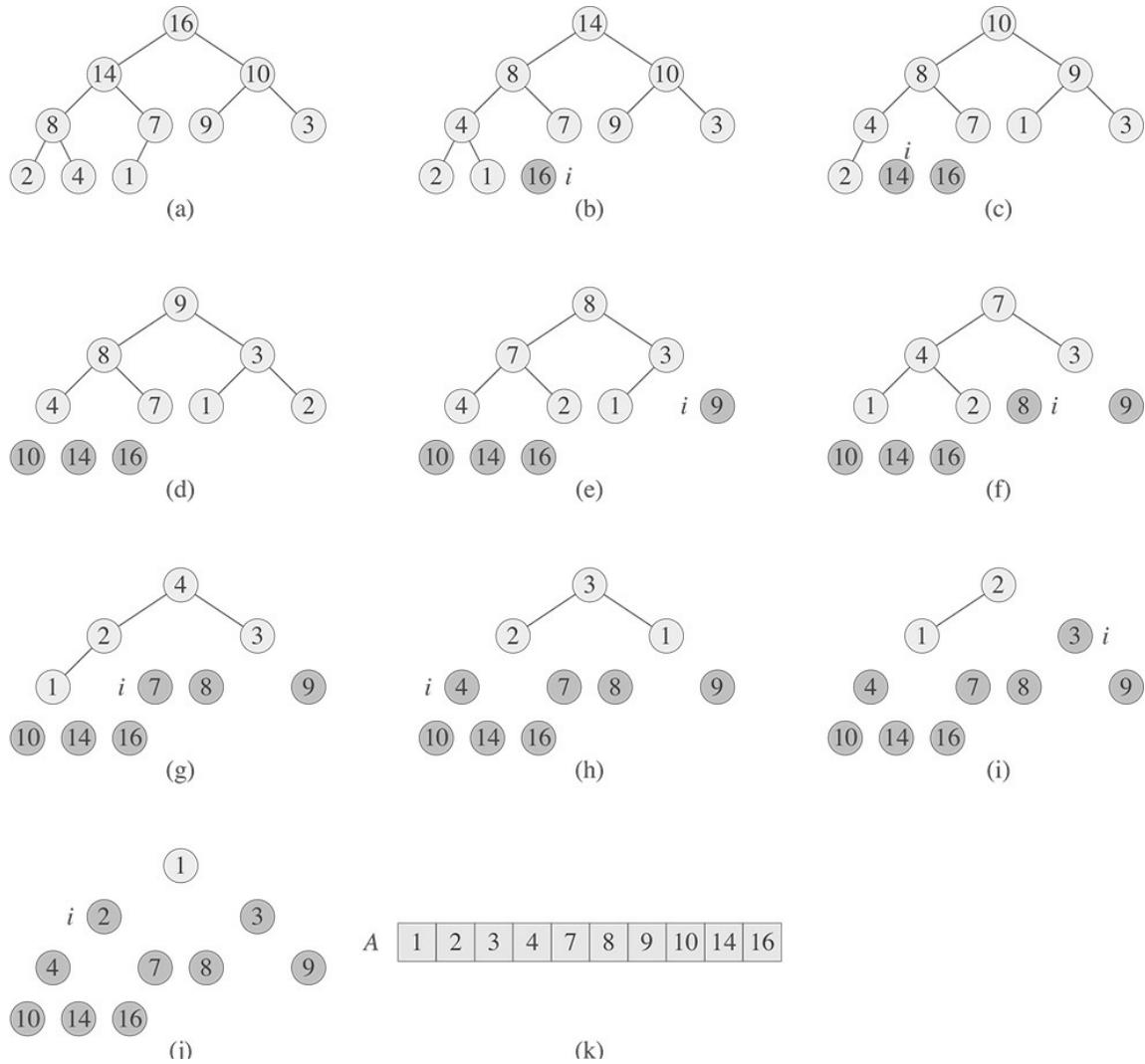
6.4 O ALGORITMO DE ORDENAÇÃO POR HEAP

O algoritmo de ordenação por heap começa usando BUILD-MAX-HEAP para construir um heap de máximo no arranjo de entrada $A[1..n]$, onde $n = A \cdot comprimento$. Visto que o elemento >máximo do arranjo está armazenado na raiz $A[1]$, podemos colocá-lo em sua posição final correta trocando-o por $A[n]$. Se agora descartarmos o nó n do heap —

e para isso basta simplesmente decrementar $A \cdot comprimento$) —, observaremos que $A[1 \mathinner{..} (n - 1)]$ pode ser facilmente transformado em um heap de máximo. Os filhos da raiz continuam sendo heaps de máximo, mas o novo elemento raiz pode violar a propriedade de heap de máximo. Porém, para restabelecer a propriedade de heap de máximo, basta chamar MAX-HEAPIFY(A, 1), que deixa um heap de máximo em $A[1 \mathinner{..} (n - 1)]$. Então, o algoritmo de ordenação por heap repete esse processo para o heap de máximo de tamanho $n - 1$ até um heap de tamanho 2. (Veja no Exercício 6.4-2 um invariante de laço preciso.)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = comprimento[A] downto 2
3     trocar A[1] com A[i]
4        A·tamanho-do-heap = A·tamanho-do-heap – 1
5        MAX-HEAPIFY(A, 1)
```

A Figura 6.4 mostra um exemplo da operação de HEAPSORT após a linha 1 ter construído o heap de máximo inicial. A figura mostra o heap de máximo antes da primeira iteração do laço **for** das linhas 2–5 e após cada iteração.

O procedimento HEAPSORT demora o tempo $O(n \lg n)$, já que a chamada a BUILD-MAX-HEAP demora o tempo $O(n)$, e cada uma das $n - 1$ chamadas a MAX-HEAPIFY demora o tempo $O(\lg n)$.

Exercícios

6.4-1 Usando a Figura 6.4 como modelo, ilustre a operação de HEAPSORT sobre o arranjo $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2 Discuta a correção de HEAPSORT usando o seguinte invariante de laço:

No início de cada iteração do laço **for** das linhas 2- 5, o subarranjo $A[1 \mathinner{..} i]$ é um heap de máximo que contém os i menores elementos de $A[1 \mathinner{..} n]$, e o subarranjo $A[i + 1 \mathinner{..} n]$ contém os $n - i$ maiores elementos de $A[1 \mathinner{..} n]$, ordenados.

6.4-3 Qual é o tempo de execução de HEAPSORT para um arranjo A de comprimento n que já está ordenado em ordem crescente? E em ordem decrescente?

Figura 6.4 Operação de HEAPSORT. (**a**) A estrutura de dados de heap de máximo logo após ter sido construída por BUILD-MAX-HEAP na linha 1. (**b**)–(**j**) O heap de máximo logo após cada chamada de MAX-HEAPIFY na linha 5, mostrando o valor de *i* nesse instante. Apenas os nós sombreados em tom mais claro permanecem no heap. (**k**) Arranjo ordenado resultante *A*.

6.4-4 Mostre que o tempo de execução do pior caso de HEAPSORT é W(n lg n).

6.4-5 ★

Mostre que, quando todos os elementos são distintos, o tempo de execução do melhor caso de HEAPSORT é W(n lg n).

6.5 FILAS DE PRIORIDADES

A ordenação por heap é um algoritmo excelente, mas uma boa implementação de quicksort, apresentada no Capítulo 7, normalmente o supera na prática. Não obstante, a estrutura de dados de heap propriamente dita tem muitas utilidades. Nesta seção, apresentaremos uma das aplicações mais populares de um heap: uma fila de prioridades eficiente. Como ocorre com os heaps, existem dois tipos de filas de prioridades: filas de prioridade máxima e filas de prioridade mínima. Focalizaremos aqui a implementação de filas de prioridade máxima, que por sua vez se baseiam em heaps de máximo; o Exercício 6.5-3 pede que você escreva os procedimentos correspondentes para filas de prioridade mínima.

Uma ***fila de prioridade*** é uma estrutura de dados para manter um conjunto S de elementos, cada qual com um valor associado denominado ***chave***. Uma ***fila de prioridade máxima*** suporta as seguintes operações:

INSERT(S, x) insere o elemento x no conjunto S. Essa operação é equivalente à operação $S = S \cup \{x\}$.

MAXIMUM(S) devolve o elemento de S que tenha a maior chave.

EXTRACT-MAX(S) remove e devolve o elemento de S que tenha a maior chave.

INCREASE-KEY(S, x, k) aumenta o valor da chave do elemento x até o novo valor k, que admite-se ser, pelo menos, tão grande quanto o valor da chave atual de x.

Entre outras aplicações, podemos usar filas de prioridade máxima para programar trabalhos em um computador compartilhado. A fila de prioridade máxima mantém o controle dos trabalhos a executar e suas prioridades relativas. Quando um trabalho termina ou é interrompido, o escalonador seleciona o trabalho de prioridade mais alta entre os trabalhos pendentes chamando EXTRACT-MAX. O escalonador pode acrescentar um novo trabalho à fila em qualquer instante chamando INSERT.

Alternativamente, uma ***fila de prioridade mínima*** suporta as operações INSERT, MINIMUM, EXTRACT-MIN e DECREASE-KEY. Uma fila de prioridade mínima pode ser usada em um simulador orientado a eventos. Os itens na fila são eventos a simular, cada qual com um instante de ocorrência associado que serve como sua chave. Os eventos devem ser simulados na ordem de seu instante de ocorrência porque a simulação de um evento pode provocar outros eventos a simular no futuro. O programa de simulação chama EXTRACT-MIN em cada etapa para escolher o próximo evento a simular. À medida que novos eventos são produzidos, o simulador os insere na fila de prioridade mínima chamando INSERT. Veremos outros usos de filas de prioridade mínima destacando a operação DECREASE-KEY, nos Capítulos 23 e 24.

Não é nenhuma surpresa que possamos usar um heap para implementar uma fila de prioridade. Em determinada aplicação, como programação de trabalhos ou simulação orientada a eventos, os elementos de uma fila de prioridade correspondem a objetos na aplicação. Muitas vezes, é necessário determinar qual objeto de aplicação corresponde a um dado elemento de fila de prioridade e vice-versa. Portanto, quando usamos um heap para implementar uma fila de prioridade, frequentemente precisamos armazenar um ***descritor*** para o objeto de aplicação correspondente em cada elemento do heap. A constituição exata do descritor (isto é, um ponteiro ou um inteiro) depende da aplicação. De modo semelhante, precisamos armazenar um descritor para o elemento do heap correspondente em cada objeto de aplicação. Nesse caso, normalmente o descritor é um índice de arranjo. Como os elementos do heap mudam de posição dentro do arranjo durante operações de heap, uma implementação real, ao reposicionar um elemento do heap, também teria de atualizar o índice do arranjo no objeto de aplicação correspondente. Visto que os detalhes de acesso a objetos de aplicação dependem muito da aplicação e de sua implementação, não os examinaremos aqui; observaremos apenas que, na prática, esses descritores precisam ser mantidos corretamente.

Agora discutiremos como implementar as operações de uma fila de prioridade máxima. O procedimento HEAP-MAXIMUM implementa a operação Maximum no tempo Q(1).

HEAP-MAXIMUM(A)
1 **return** $A[1]$

O procedimento HEAP-EXTRACT-MAX implementa a operação EXTRACT-MAX. Ele é semelhante ao corpo do laço **for** (linhas 3-5) do procedimento HEAPSORT.

HEAP-EXTRACT-MAX(A)

1 **if** $A \cdot tamanho\text{-}do\text{-}heap < 1$
2 **error** "heap underflow"
3 $max = A[1]$
4 $A[1] = A[A \cdot tamanho\text{-}do\text{-}heap]$
5 $A \cdot tamanho\text{-}do\text{-}heap = A \cdot tamanho\text{-}do\text{-}heap - 1$
6 MAX-HEAPIFY(A, 1)
7 **return** max

O tempo de execução de HEAP-EXTRACT-MAX é $O(\lg n)$, já que ele executa apenas uma quantidade constante de trabalho além do tempo $O(\lg n)$ para MAX-HEAPIFY.

O procedimento HEAP-INCREASE-KEY implementa a operação INCREASE-KEY. Um índice i para o arranjo identifica o elemento da fila de prioridade cuja chave queremos aumentar. Primeiro, o procedimento atualiza a chave do elemento $A[i]$ para seu novo valor. Visto que aumentar a chave de $A[i]$ pode violar a propriedade de heap de máximo, o procedimento, de um modo que lembra o laço de inserção (linhas 5-7) de INSERTION-SORT da Seção 2.1, percorre um caminho simples desde esse nó até a raiz para encontrar um lugar adequado para a chave recém-aumentada. Enquanto HEAP-INCREASE-KEY percorre esse caminho, compara repetidamente um elemento a seu pai, permutando suas chaves, prossegue se a chave do elemento for maior e termina se a chave do elemento for menor, visto que a propriedade de heap de máximo agora é válida. (Veja no Exercício 6.5-5 um invariante de laço preciso.)

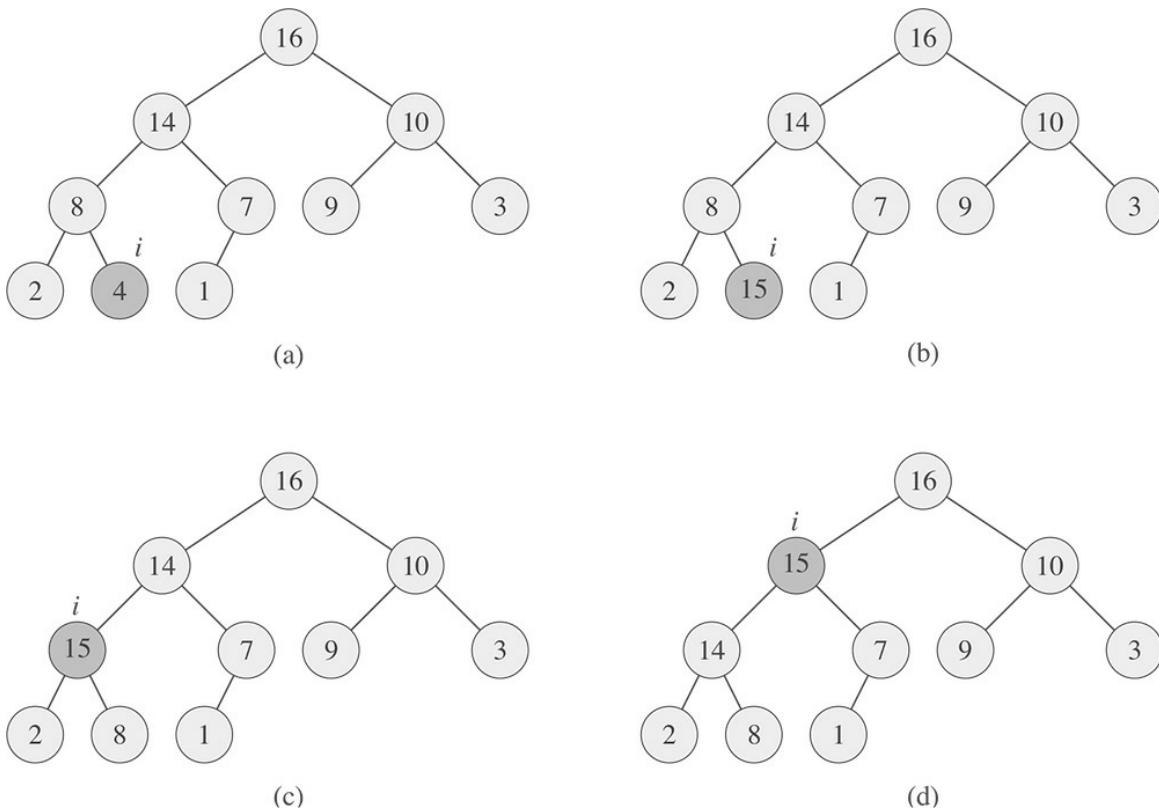
HEAP-INCREASE-KEY(A, i, $chave$)

1 **if** $chave < A[i]$
2 **error** "nova chave é menor que chave atual"
3 $A[i] = chave$
4 **while** $i > 1$ e $A[\text{PARENT}(i)] < A[i]$
5 troca $A[i]$ com $A[\text{PARENT}(i)]$
6 $i = \text{PARENT}(i)$

A Figura 6.5 mostra um exemplo de operação HEAP-INCREASE-KEY. O tempo de execução de HEAP-INCREASE-KEY para um heap de n elementos é $O(\lg n)$, visto que o caminho traçado desde o nó atualizado na linha 3 até a raiz tem comprimento $O(\lg n)$.

O procedimento MAX-HEAP-INSERT implementa a operação INSERT. Toma como entrada a chave do novo elemento a ser inserido no heap de máximo A. Primeiro, o procedimento expande o heap de máximo, acrescentando à árvore uma nova folha cuja chave é $-\infty$. Em seguida, chama HEAP-INCREASE-KEY para ajustar a chave desse novo nó em seu valor correto e manter a propriedade de heap de máximo.

MAX-HEAP-INSERT(A, $chave$)

1 $A \cdot tamanho\text{-}do\text{-}heap = A \cdot tamanho\text{-}do\text{-}heap + 1$
2 $A[A \cdot tamanho\text{-}do\text{-}heap] = -\infty$
3 HEAP-INCREASE-KEY(A, $A \cdot tamanho\text{-}do\text{-}heap$, $chave$)

O tempo de execução de MAX-HEAP-INSERT para um heap de n elementos é $O(\lg n)$.

Resumindo, um heap pode suportar qualquer operação de fila de prioridade em um conjunto de tamanho n no tempo $O(\lg n)$.

Exercícios

6.5-1 Ilustre a operação de HEAP-EXTRACT-MAX sobre o heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$.

6.5-2 Ilustre a operação de MAX-HEAP-INSERT(A, 10) sobre o heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$.

6.5-3 Escreva pseudocódigos para os procedimentos HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY e MIN-HEAP-INSERT que implementem uma fila de prioridade mínima com um heap de mínimo.

6.5-4 Por que nos preocupamos em definir a chave do nó inserido como $-\infty$ na linha 2 de MAX-HEAP-INSERT quando a nossa próxima ação é aumentar sua chave para o valor desejado?

6.5-5 Demonstre a correção de HEAP-INCREASE-KEY usando o seguinte invariante de laço:

No início de cada iteração do laço **while** das linhas 4–6, o subarranjo $A[1 \,..\, A \cdot \textit{tamanho-do-heap}]$ satisfaz a propriedade de heap de máximo, exceto que pode haver uma violação: $A[i]$ pode ser maior que $A[\text{PARENT}(i)]$.

Figura 6.5 Operação de HEAP-INCREASE-KEY. (**a**) O heap de máximo da Figura 6.4(a) com um nó cujo índice é *i* em sombreado de tom mais escuro. (**b**) A chave desse nó é aumentada para 15. (**c**) Depois de uma iteração do laço **while** das linhas 4–6, o nó e seu pai trocaram chaves, e o índice *i* sobe para o pai. **(d)** Heap de máximo após mais uma iteração do laço **while**. Nesse ponto, $A[\text{PARENT}(i)] \geq A[i]$. Agora, a propriedade de heap de máximo é válida e o procedimento termina.

Você pode supor que o subarranjo A[1 .. *A · tamanho-do-heap]* satisfaz a propriedade de heap de máximo no instante em que HEAP-INCREASE-KEY é chamado.

6.5-6 Cada operação de troca na linha 5 de HEAP-INCREASE-KEY normalmente, requer três atribuições. Mostre como usar a ideia do laço interno de INSERTION-SORT para reduzir as três atribuições a apenas uma atribuição.

6.5-7 Mostre como implementar uma fila primeiro a entrar, primeiro a sair com uma fila de prioridade. Mostre como implementar uma pilha com uma fila de prioridade. (Filas e pilhas são definidas na Seção 10.1.)

6.5-8 A operação HEAP-DELETE(A, i) elimina o item no nó i do heap A. Dê uma implementação de HEAP-DELETE que seja executada no tempo $O(\lg n)$ para um heap de máximo de n elementos.

6.5-9 Dê um algoritmo de tempo $O(n \lg k)$ para intercalar k listas ordenadas em uma única lista ordenada, onde n é o número total de elementos em todas as listas de entrada. (*Sugestão*: Use um heap de mínimo para fazer a intercalação de k entradas.)

Problemas

6-1 Construir um heap com a utilização de inserção

Podemos construir um heap chamando repetidamente MAX-HEAP-INSERT para inserir os elementos no heap. Considere a seguinte variação do procedimento BUILD-MAX-HEAP:

BUILD-MAX-HEAP$'$(A)
1 $A{\cdot}tamanho\text{-}do\text{-}heap = 1$
2 **for** $i = 2$ **to** $A{\cdot}comprimento$
3 MAX-HEAP-INSERT(A, $A[i]$)

a. Os procedimentos BUILD-MAX-HEAP e BUILD-MAX-HEAP$'$ sempre criam o mesmo heap quando são executados sobre o mesmo arranjo de entrada? Prove que isso ocorre ou, então, dê um contraexemplo.

b. Mostre que, no pior caso, BUILD-MAX-HEAP$'$ requer o tempo Q(n lg n) para construir um heap de n elementos.

6-2 Análise de heaps d-ários

Um ***heap d-ário*** é semelhante a um heap binário, mas (com uma única exceção possível) nós que não são folhas têm d filhos em vez de dois filhos.

a. Como você representaria um heap d-ário em um arranjo?

b. Qual é a altura de um heap d-ário de n elementos em termos de n e d?

c. Dê uma implementação eficiente de EXTRACT-MAX em um heap de máximo d-ário. Analise seu tempo de execução em termos de d e n.

d. Dê uma implementação eficiente de INSERT em um heap de máximo d-ário. Analise seu tempo de execução em termos de d e n.

e. Dê uma implementação eficiente de INCREASE-KEY(A, i, k), que sinaliza um erro se $k < A[i]$ mas, caso contrário, ajusta $A[i] = k$ e então atualiza adequadamente a estrutura do heap de máximo d-ário. Analise seu tempo de execução em termos de d e n.

6-3 Quadros de Young

Um ***quadro de Young*** $m \times n$ é uma matriz $m \times n$, tal que as entradas de cada linha estão em sequência ordenada da esquerda para a direita, e as entradas de cada coluna estão em sequência ordenada de cima para baixo. Algumas das entradas de um quadro de Young podem ser ∞, que tratamos como elementos inexistentes. Assim, um quadro de Young pode ser usado para conter $r \leq mn$ números finitos.

a. Trace um quadro de Young 4×4 contendo os elementos $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Demonstre que um quadro de Young $m \times n$ Y é vazio se $Y[1, 1] = \infty$. Demonstre que Y é cheio (contém mn elementos) se $Y[m, n] < \infty$.

c. Dê um algoritmo para implementar EXTRACT-MIN em um quadro de Young $m \times n$ não vazio que é executado no tempo $O(m + n)$. Seu algoritmo deve usar uma subrotina recursiva que resolve um problema $m \times n$ resolvendo recursivamente um subproblema $(m - 1) \times n$ ou um subproblema $m \times (n - 1)$. (*Sugestão*: Pense em MAX-HEAPIFY.) Defina $T(p)$, onde $p = m + n$, como o tempo de execução máximo de EXTRACT-MIN em qualquer quadro de Young $m \times n$. Dê e resolva uma recorrência para $T(p)$ que produza o limite de tempo $O(m + n)$.

d. Mostre como inserir um novo elemento em um quadro de Young $m \times n$ não cheio no tempo $O(m + n)$.

e. Sem usar nenhum outro método de ordenação como subrotina, mostre como utilizar um quadro de Young $n \times n$ para ordenar n^2 números no tempo $O(n^3)$.

f. Dê um algoritmo de tempo $O(m + n)$ para determinar se um dado número está armazenado em determinado quadro de Young $m \times n$.

NOTAS DO CAPÍTULO

O algoritmo de ordenação por heap foi criado por Williams [357], que também descreveu como implementar uma fila de prioridades com um heap. O procedimento BUILD-MAX-HEAP foi sugerido por Floyd [106].

Usamos heaps de mínimo para implementar filas de prioridade mínima nos Capítulos 16, 23 e 24. Também damos uma implementação com limites de tempo melhorados para certas operações no Capítulo 19 e, considerando que as chaves são escolhidas de um conjunto limitado de inteiros não negativos, no Capítulo 20.

Quando os dados são inteiros de b bits e a memória do computador consiste em palavras endereçáveis de b bits, Fredman e Willard [115] mostraram como implementar MINIMUM no tempo $O(1)$ e INSERT e EXTRACT-MIN no tempo $O(\sqrt{\lg n})$. Thorup [337] melhorou o limite $O(\sqrt{\lg n})$ para o tempo $O(\sqrt{\lg n})$. Esse limite usa quantidade de espaço ilimitada em n, mas pode ser implementado em espaço linear com a utilização de hashing aleatorizado.

Um caso especial importante de filas de prioridades ocorre quando a sequência de operações de EXTRACT-MIN é ***monotônica,*** isto é, os valores retornados por operações sucessivas de EXTRACT-MIN são monotonicamente crescentes com o tempo. Esse caso surge em várias aplicações importantes, como o algoritmo de caminhos mais curtos de fonte única de Dijkstra, que discutiremos no Capítulo 24, e na simulação de eventos discretos. Para o algoritmo de Dijkstra, é particularmente importante que a operação DECREASE-KEY seja implementada eficientemente.

No caso monotônico, se os dados são inteiros na faixa 1, 2, ..., C, Ahuja, Melhorn, Orlin e Tarjan [8] descrevem como implementar EXTRACT-MIN e INSERT no tempo amortizado $O(\lg C)$ (consulte o Capítulo 17 para obter mais informações sobre análise amortizada) e DECREASE-KEY no tempo $O(1)$, usando uma estrutura de dados denominada heap digital. O limite $O(\lg C)$ pode ser melhorado para $O(\sqrt{\lg C})$ com a utilização de heaps de Fibonacci (consulte o Capítulo 19) em conjunto com heaps digitais. Cherkassky, Goldberg e Silverstein [65] melhoraram ainda mais o limite até o tempo esperado $O(\lg^{1/3 +} C)$ combinando a estrutura de baldes em vários níveis de Denardo e Fox [85] com o heap de Thorup já mencionado. Raman [291] aprimorou mais ainda esses resultados para obter um limite de $O(\min(\lg^{1/4 +} C, \lg^{1/3 +} n))$, para qualquer > 0 fixo.

7 QUICKSORT

O algoritmo quicksort (ordenação rápida) tem tempo de execução do pior caso de Q(n_2) para um arranjo de entrada de n números. Apesar desse tempo de execução lento para o pior caso, muitas vezes, o quicksort é a melhor opção prática para ordenação, devido à sua notável eficiência na média: seu tempo de execução esperado é Q(n lg n), e os fatores constantes ocultos na notação Q(n lg n) são bastante pequenos. Ele também apresenta a vantagem de ordenar no lugar (veja página 17) e funciona bem até mesmo em ambientes de memória virtual.

A Seção 7.1 descreve o algoritmo e uma subrotina importante usada pelo quicksort para particionamento. Como o comportamento do quicksort é complexo, começaremos com uma discussão intuitiva de seu desempenho na Seção 7.2 e adiaremos sua análise precisa até o final do capítulo. A Seção 7.3 apresenta uma versão de quicksort que utiliza amostragem aleatória. Esse algoritmo tem um bom tempo de execução esperado, e nenhuma entrada específica induz seu comportamento do pior caso. A Seção 7.4 analisa o algoritmo aleatorizado, mostrando que ele é executado no tempo Q (n_2) no pior caso e, considerando elementos distintos, no tempo esperado $O(n \lg n)$.

7.1 DESCRIÇÃO DO QUICKSORT

O quicksort, como a ordenação por intercalação, aplica o paradigma de divisão e conquista introduzido na Seção 2.3.1. Descrevemos a seguir, o processo de três etapas do método de divisão e conquista para ordenar um subarranjo típico $A[p \,..\, r]$.

Divisão: Particionar (reorganizar) o arranjo $A[p \,..\, r]$ em dois subarranjos (possivelmente vazios) $A[p \,..\, q - 1]$ e $A[q + 1 \,..\, r]$ tais que, cada elemento de $A[p \,..\, q - 1]$ é menor ou igual a $A[q]$ que, por sua vez, é menor ou igual a cada elemento de $A[q + 1 \,..\, r]$. Calcular o índice q como parte desse procedimento de particionamento.

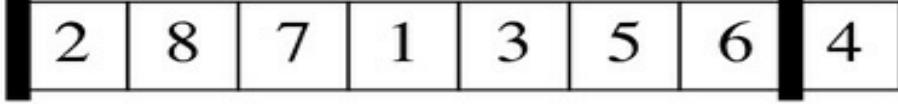
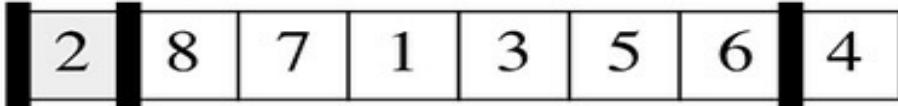
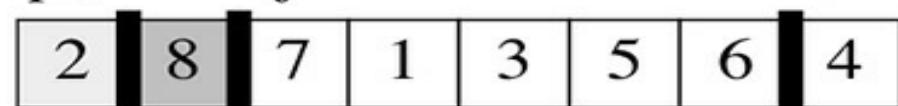
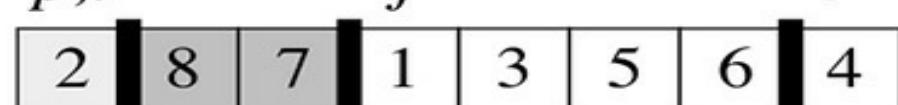
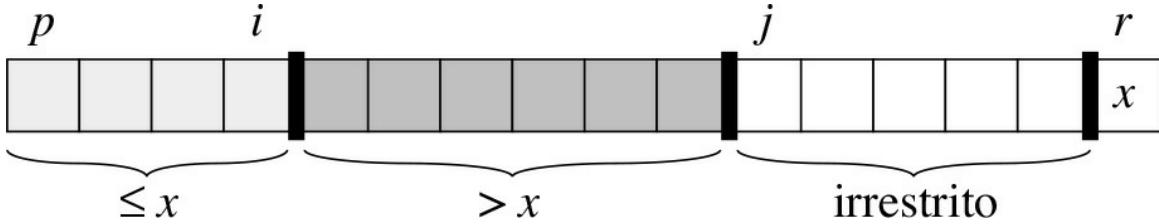
Conquista: Ordenar os dois subarranjos $A[p \,..\, q - 1]$ e $A[q + 1 \,..\, r]$ por chamadas recursivas a quicksort.

Combinação: Como os subarranjos já estão ordenados, não é necessário nenhum trabalho para combiná-los: o arranjo $A[p \,..\, r]$ inteiro agora está ordenado.

O seguinte procedimento implementa o quicksort:

```
QUICKSORT(A, p, r)
1  if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

Para ordenar um arranjo A inteiro, a chamada inicial é QUICKSORT(A, 1, $A \cdot$ *comprimento*).

Particionamento do arranjo

A chave para o algoritmo é o procedimento PARTITION, que reorganiza o subarranjo $A[p \, .. \, r]$ no lugar.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            trocar A[i] por A[j]
7  trocar A[i + 1] por A[r]
8  return i + 1
```

A Figura 7.1 mostra como PARTITION funciona para um arranjo de oito elementos. PARTI- TION sempre seleciona um elemento $x = A[r]$ como um elemento ***pivô*** ao redor do qual particionar o subarranjo $A[p \, .. \, r]$. À medida que é executado, o procedimento reparte o arranjo em quatro regiões (possivelmente vazias). No início de cada iteração do laço **for** nas linhas 3−6, as regiões satisfazem certas propriedades, mostradas na Figura 7.2. Enunciamos essas propriedades como um invariante de laço:

Figura 7.1 A operação PARTITION para uma amostra de arranjo. A entrada $A[r]$ do arranjo torna-se o elemento pivô x. Os elementos do arranjo sombreados em tom mais claro estão na primeira partição com valores não maiores que x. Elementos sombreados em tom mais escuro estão na segunda partição com valores maiores que x. Os elementos não sombreados ainda não foram inseridos em uma das duas primeiras partições, e o elemento final em fundo branco é o pivô x. (**a**) Configuração inicial do arranjo e das variáveis. Nenhum dos elementos foi inserido em nenhuma das duas primeiras partições. (**b**) O valor 2 é "permutado por ele mesmo" e inserido na partição de valores menores. (**c**)–(**d**) Os valores 8 e 7 são acrescentados à partição de valores maiores. (**e**) Os valores 1 e 8 são permutados, e a partição menor cresce. (**f**) Os valores 3 e 7 são permutados, e a partição menor cresce. (**g**)–(**h**) A partição maior cresce para incluir 5 e 6, e o laço termina. (**i**) Nas linhas 7–8, o elemento pivô é permutado de modo que se encontra entre as duas partições.

Figura 7.2 As quatro regiões mantidas pelo procedimento PARTITION em um subarranjo $A[p .. r]$. Os valores em $A[p .. i]$ são menores ou iguais a x, os valores em $A[i + 1 .. j - 1]$ são maiores que x e $A[r] = x$. O subarranjo $A[j .. r - 1]$ pode aceitar quaisquer valores.

No início de cada iteração do laço das linhas 3–6, para qualquer índice k do arranjo,

1. Se $p \leq k \leq i$, então $A[k] \leq x$.
2. Se $i + 1 \leq k \leq j - 1$, então $A[k] > x$.
3. Se $k = r$, então $A[k] = x$.

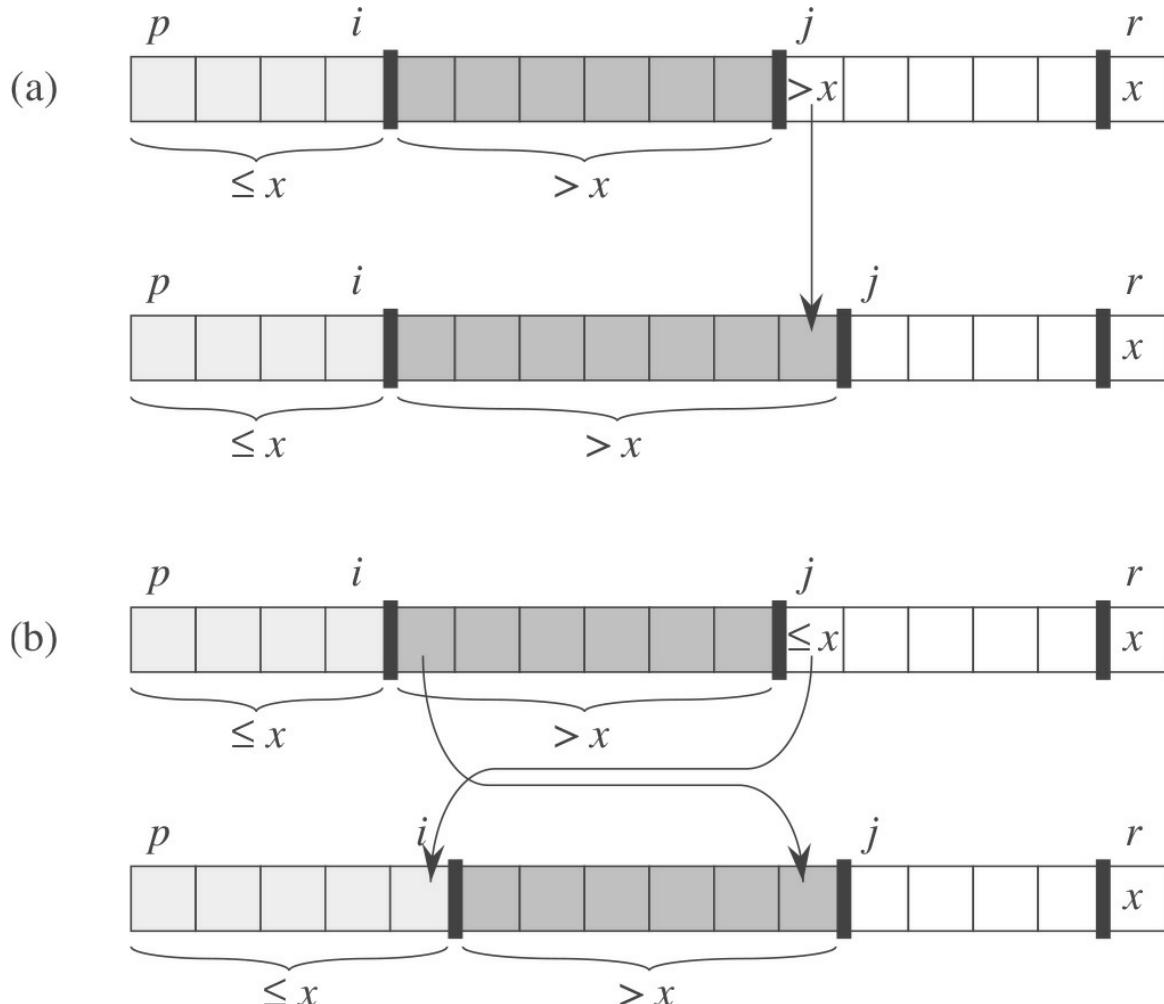
Os índices entre j e $r - 1$ não são abrangidos por nenhum dos três casos, e os valores nessas entradas não têm nenhuma relação particular com o pivô x.

Precisamos mostrar que esse invariante de laço é verdadeiro antes da primeira iteração, que cada iteração do laço mantém o invariante e que o invariante fornece uma propriedade útil para mostrar correção quando o laço termina.

Inicialização: Antes da primeira iteração do laço, $i = p - 1$ e $j = p$. Como não há nenhum valor entre p e i e nenhum valor entre $i + 1$ e $j - 1$, as duas primeiras condições do invariante de laço são satisfeitas trivialmente. A atribuição na linha 1 satisfaz a terceira condição.

Manutenção: Como mostra a Figura 7.3, consideraremos dois casos, dependendo do resultado do teste na linha 4. A Figura 7.3(a) mostra o que acontece quando $A[j] > x$; a única ação no laço é incrementar j. Depois que j é incrementado, a condição 2 é válida para $A[j - 1]$ e todas as outras entradas permanecem inalteradas.

A Figura 7.3(b) mostra o que acontece quando $A[j] \leq x$; o laço incrementa i, permuta $A[i]$ e $A[j]$, e então incrementa j. Por causa da troca, agora temos que $A[i] \leq x$, e a condição 1 é satisfeita. De modo semelhante, também temos que $A[j - 1] > x$, visto que o item que foi permutado para dentro de $A[j - 1]$ é, pelo invariante de laço, maior que x.

Término: No término, $j = r$. Portanto, toda entrada no arranjo está em um dos três conjuntos descritos pelo invariante, e particionamos os valores no arranjo em três conjuntos: os menores ou iguais a x, os maiores que x e um conjunto unitário contendo x.

As duas linhas finais de PARTITION terminam permutando o elemento pivô pelo elemento maior que x na extrema esquerda e, com isso, deslocam o pivô até seu lugar correto no arranjo particionado; em seguida, retornam o novo índice do pivô. Agora, a saída de PARTITION satisfaz as especificações dadas para a etapa dividir. Na verdade, ela satisfaz uma condição ligeiramente mais forte: após a linha 2 de QUICKSORT, $A[q]$ é estritamente menor do que todo elemento de $A[q + 1 . . r]$.

O tempo de execução de PARTITION para o subarranjo $A[p .. r]$ é Q(n), onde $n = r - p + 1$ (veja Exercício 7.1-3).

Exercícios

7.1-1 Usando a Figura 7. 1 como modelo, ilustre a operação de PARTITION sobre o arranjo $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

7.1-2 Qual valor de q PARTITION retorna quando todos os elementos no arranjo $A[p .. r]$ têm o mesmo valor? Modifique PARTITION de modo que $q = (p + r)/2$ quando todos os elementos no arranjo $A[p .. r]$ têm o mesmo valor.

Figura 7.3 Os dois casos para uma iteração do procedimento PARTITION. (**a**) Se $A[j] > x$, a única ação é incrementar j, que mantém o invariante de laço. (**b**) Se $A[j] \leq x$, o índice i é incrementado, $A[i]$ e $A[j]$ são permutados e, então, j é incrementado. Novamente, o invariante de laço é mantido.

7.1-3 Apresente um breve argumento mostrando que o tempo de execução de PARTITION para um subarranjo de tamanho n é Q(n).

7.1-4 Como você modificaria QUICKSORT para ordenar em ordem não crescente?

7.2 O DESEMPENHO DO QUICKSORT

O tempo de execução do quicksort depende de o particionamento ser balanceado ou não balanceado, o que por sua vez depende de quais elementos são usados para particionar. Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto a ordenação por intercalação. Contudo, se o particionamento é não balanceado, ele pode ser executado assintoticamente tão lento quanto a ordenação por inserção. Nesta seção, investigaremos informalmente como o quicksort se comporta sob as premissas do particionamento balanceado e do particionamento não balanceado.

Particionamento no pior caso

O comportamento do pior caso para o quicksort ocorre quando a rotina de particionamento produz um subproblema com $n - 1$ elementos e um com 0 elementos. (Provamos essa afirmativa na Seção 7.4.1.) Vamos considerar que esse particionamento não balanceado surja em cada chamada recursiva. O particionamento custa o tempo Q(n). Visto que a chamada recursiva para um arranjo de tamanho 0 apenas retorna, $T(0)$ = Q(1) e a recorrência para o tempo de execução é

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n). \end{aligned}$$

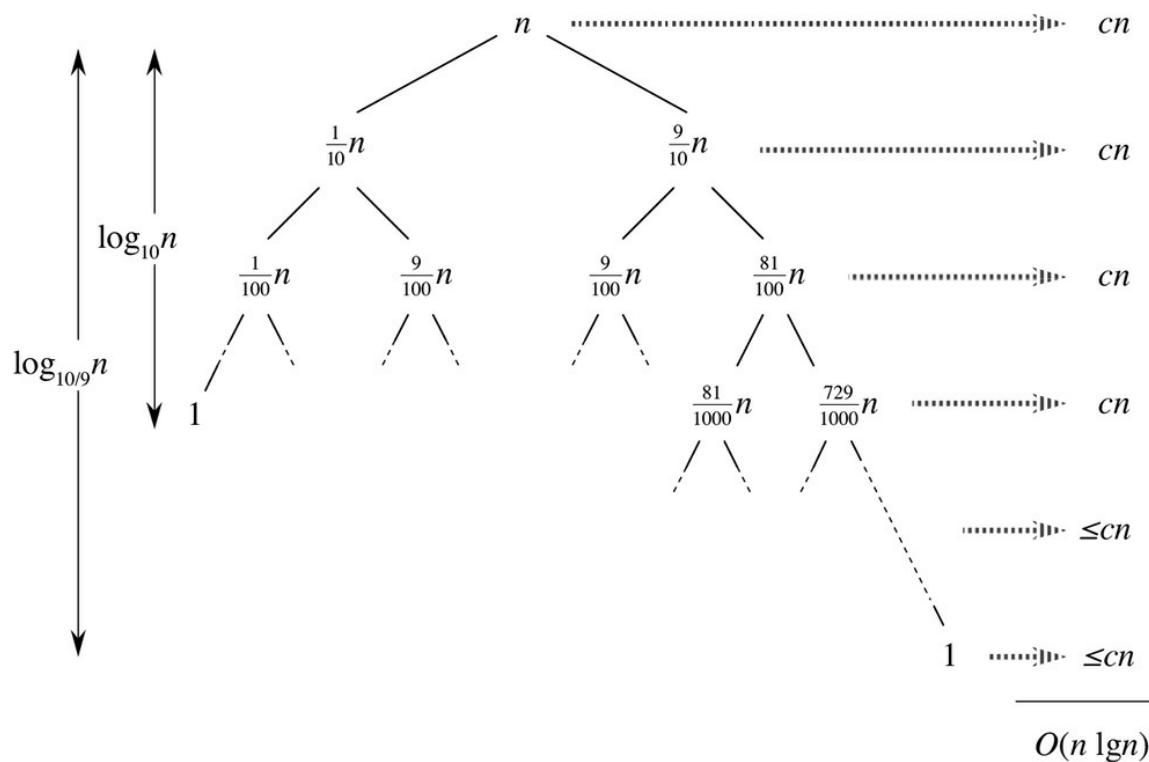
Intuitivamente, se somarmos os custos incorridos em cada nível da recursão, obteremos uma série aritmética (equação (A.2)), cujo valor chega a Q(n_2). Na realidade, é simples usar o método de substituição para provar que a recorrência $T(n) = T(n-1)$ + Q(n) tem a solução $T(n)$ = Q(n_2). (Veja o Exercício 7.2-1.)

Assim, se o particionamento é maximamente não balanceado em todo nível recursivo do algoritmo, o tempo de execução é Q(n_2). Portanto, o tempo de execução do pior caso do quicksort não é melhor que o da ordenação por inserção. Além disso, o tempo de execução Q(n_2) ocorre quando o arranjo de entrada já está completamente ordenado — uma situação comum na qual a ordenação por inserção é executada no tempo $O(n)$.

Particionamento do melhor caso

Na divisão mais equitativa possível, PARTITION produz dois subproblemas, cada um de tamanho não maior que $n/2$, já que um é de tamanho $n/2$ e o outro é de tamanho $n/2 - 1$. Nesse caso, a execução do quicksort é muito mais rápida. Então, a recorrência para o tempo de execução é

$$T(n) = 2T(n/2) + \Theta(n),$$

onde toleramos o desleixo de ignorar o piso e o teto e de subtrair 1. Pelo caso 2 do teorema mestre (Teorema 4.1), a solução dessa recorrência é $T(n)$ = Q(n lg n). Balanceando igualmente os dois lados da partição em todo nível da recursão, obtemos um algoritmo assintoticamente mais rápido.

Particionamento balanceado

O tempo de execução do caso médio do quicksort é muito mais próximo do melhor caso que do pior caso, como mostrarão as análises da Seção 7.4. A chave para entender por que é entender como o equilíbrio do particionamento é refletido na recorrência que descreve o tempo de execução.

Por exemplo, suponha que o algoritmo de particionamento sempre produza uma divisão proporcional de 9 para 1, que à primeira vista parece bastante desequilibrada. Então, obtemos a recorrência no tempo de execução do quicksort,

e incluímos explicitamente a constante c oculta no termo Q(n). A Figura 7.4 mostra a árvore de recursão para essa recorrência. Note que todo nível da árvore tem custo cn até a recursão alcançar uma condição de contorno à profundidade $\log_{10} n$ = Q(lg n); daí em diante, os níveis têm no máximo o custo cn. A recursão termina na profundidade $\log_{10/9} n$ = Q(lg n). Portanto, o custo total do quicksort é $O(n \lg n)$. Assim, com uma divisão na proporção de 9 para 1 em todo nível de recursão, o que intuitivamente parece bastante desequilibrado, o quicksort é executado no tempo $O(n \lg n)$ — assintoticamente, o mesmo tempo que teríamos se a divisão fosse exatamente ao meio. De fato, até mesmo uma divisão de 99 para 1 produz um tempo de execução $O(n \lg n)$. Na verdade, qualquer divisão de proporcionalidade *constante* produz uma árvore de recursão de profundidade Q(lg n), em que o custo em cada nível é $O(n)$. Portanto, o tempo de execução é Q(n lg n) sempre que a divisão tiver proporcionalidade constante.

$$T(n) = T(9n/10) + T(n/10) + cn,$$

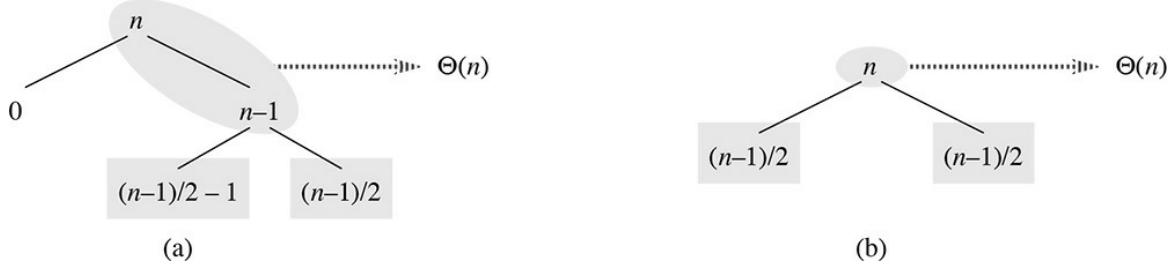



Figura 7.4 Uma árvore de recursão para Quicksort, na qual Partition sempre produz uma divisão de 9 para 1, resultando no tempo de execução $O(n \lg n)$. Os nós mostram tamanhos de subproblemas, com custos por nível à direita. Os custos por nível incluem a constante c implícita no termo Q(n).

Intuição para o caso médio

Para desenvolver uma noção clara do comportamento aleatorizado do quicksort, temos de adotar uma premissa em relação à frequência com que esperamos encontrar as várias entradas. O comportamento do quicksort depende da ordenação relativa dos valores nos elementos do arranjo dados como entrada, e não dos valores particulares no arranjo. Como em nossa análise probabilística do problema da contratação na Seção 5.2, suporemos por enquanto que todas as permutações dos números de entrada são igualmente prováveis.

Quando executamos o quicksort sobre um arranjo de entrada aleatório, é muito improvável que o particionamento ocorra do mesmo modo em todo nível, como nossa análise informal pressupôs. Esperamos que algumas divisões serão razoavelmente bem equilibradas e outras serão razoavelmente desequilibradas. Por exemplo, o Exercício 7.2-6 pede para você mostrar que, em aproximadamente 80% do tempo, Partition produz uma divisão mais equilibrada que 9 para 1, e em aproximadamente 20% do tempo ele produz uma divisão menos equilibrada que 9 para 1.

No caso médio, Partition produz um misto de divisões "boas" e "ruins". Em uma árvore de recursão para uma execução de Partition para o caso médio, as divisões boas e ruins estão distribuídas aleatoriamente por toda a árvore. Suponha, por intuição, que as divisões boas e ruins se alternem nos níveis da árvore, que as divisões boas sejam divisões do melhor caso e as divisões ruins sejam divisões do pior caso. A Figura 7.5(a) mostra as divisões em dois níveis consecutivos na árvore de recursão. Na raiz da árvore, o custo é n para particionamento e os subarranjos produzidos têm tamanhos $n - 1$ e 0: o pior caso. No nível seguinte, o subarranjo de tamanho $n - 1$ sofre particionamento do melhor caso e subdivide-se em dois subarranjos de tamanhos $(n - 1)/2 - 1$ e $(n - 1)/2$. Vamos supor que o custo da condição de contorno é 1 para o subarranjo de tamanho 0.

Figura 7.5 (a) Dois níveis de uma árvore de recursão para quicksort. O particionamento na raiz custa n e produz uma divisão "ruim": dois subarranjos de tamanhos 0 e $n - 1$. O particionamento do subarranjo de tamanho $n - 1$ custa $n - 1$ e produz uma divisão "boa": subarranjos de tamanhos $(n - 1)/2 - 1$ e $(n - 1)/2$. **(b)** Um único nível de uma árvore de recursão que está muito bem equilibrada. Em ambas as partes, o custo de particionamento para os subproblemas mostrados nas elipses sombreadas é $Q(n)$. Ainda assim, os subproblemas que ainda faltam resolver em (a), mostrados nos retângulos sombreados, não são maiores que os subproblemas que ainda faltam resolver em (b).

A combinação da divisão ruim seguida pela divisão boa produz três subarranjos de tamanhos 0, $(n - 1)/2 - 1$ e $(n - 1)/2$, a um custo de particionamento combinado de $Q(n) + Q(n - 1) = Q(n)$. Certamente essa situação não é pior que a da Figura 7.5(b), ou seja, um único nível de particionamento que produz dois subarranjos de tamanho $(n - 1)/2$, ao custo $Q(n)$. Ainda assim, esta última situação é equilibrada! Intuitivamente, o custo $Q(n - 1)$ da divisão ruim pode ser absorvido no custo $Q(n)$ da divisão boa, e a divisão resultante é boa. Assim, o tempo de execução do quicksort, quando os níveis se alternam entre divisões boas e ruins, é semelhante ao tempo de execução para divisões boas sozinhas: ainda $O(n \lg n)$, mas com uma constante ligeiramente maior oculta pela notação O. Faremos uma análise rigorosa do tempo de execução esperado de uma versão aleatorizada do quicksort na Seção 7.4.2.

Exercícios

7.2-1 Use o método de substituição para provar que a recorrência $T(n) = T(n - 1) + Q(n)$ tem a solução $T(n) = Q(n_2)$, como afirmamos no início da Seção 7.2.

7.2-2 Qual é o tempo de execução de Quicksort quando todos os elementos do arranjo A têm o mesmo valor?

7.2-3 Mostre que o tempo de execução do Quicksort é $Q(n_2)$ quando o arranjo A contém elementos distintos e está ordenado em ordem decrescente.

7.2-4 Os bancos frequentemente registram transações em uma conta na ordem dos horários das transações, mas muitos clientes gostam de receber em seus extratos bancários uma relação de cheques por ordem do número do cheque. Normalmente, as pessoas preenchem cheques na ordem do número do cheque, e os comerciantes normalmente, os descontam com presteza razoável. Portanto, o problema de converter a ordenação pela hora da transação na ordenação pelo número do cheque é o problema de ordenar uma entrada quase ordenada.

Demonstre que o procedimento INSERTION-SORT tenderia a superar o procedimento QUICKSORT nesse problema.

7.2-5 Suponha que as divisões em todo nível do quicksort estejam na proporção $1 - \alpha$ para α, onde $0 < \alpha \leq 1/2$ é uma constante. Mostre que a profundidade mínima de uma folha na árvore de recursão é aproximadamente $-\lg n/\lg \alpha$ e a profundidade máxima é aproximadamente $-\lg n/\lg(1 - \alpha)$. (Não se preocupe com arredondamento.)

7.2-6 ★

Demonstre que, para qualquer constante $0 < \alpha \leq 1/2$, a probabilidade de que, em um arranjo de entradas aleatórias, PARTITION produza uma divisão mais equilibrada que $1 - \alpha$ para α é aproximadamente $1 - 2\alpha$.

7.3 UMA VERSÃO ALEATORIZADA DO QUICKSORT

Quando exploramos o comportamento do caso médio do quicksort, adotamos a premissa de que todas as permutações dos números de entrada são igualmente prováveis. Porém, em uma situação de engenharia nem sempre podemos esperar que tal premissa se mantenha válida (veja o Exercício 7.2-4). Como vimos na Seção 5.3, às vezes, podemos acrescentar aleatorização a um algoritmo para obter bom desempenho esperado para todas as entradas. Muitos consideram a versão aleatorizada do quicksort resultante o algoritmo de ordenação preferido para entradas suficientemente grandes.

Na Seção 5.3, aleatorizamos nosso algoritmo permutando explicitamente a entrada. Também poderíamos fazer isso para o quicksort, mas uma técnica de aleatorização diferente, denominada ***amostragem aleatória***, produz uma análise mais simples. Em vez de sempre usar $A[r]$ como pivô, selecionaremos um elemento escolhido aleatorimente no subarranjo $A[p \mathinner{..} r]$. Para tal, em primeiro lugar permutamos o elemento $A[r]$ por um elemento escolhido aleatoriamente em $A[p \mathinner{..} r]$. Tomar amostras aleatórias na faixa $p, ..., r$, assegura que o elemento pivô $x = A[r]$ tem a mesma probabilidade de ser qualquer um dos $r - p + 1$ elementos no subarranjo. Como escolhemos o elemento pivô aleatoriamente, esperamos que a divisão do arranjo de entrada seja razoavelmente bem equilibrada na média.

As mudanças em PARTITION e QUICKSORT são pequenas. No novo procedimento de partição, simplesmente implementamos a troca antes do particionamento propriamente dito:

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  trocar A[p] por A[i]
3  return PARTITION(A, p, r)
```

O novo quicksort chama RANDOMIZED-PARTITION em vez de PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Analisaremos esse algoritmo na próxima seção.

Exercícios

7.3-1 Por que analisamos o tempo de execução esperado de um algoritmo aleatorizado e não seu tempo de execução do pior caso?

7.3-2 Durante a execução do procedimento RANDOMIZED-QUICKSORT, quantas chamadas são feitas ao gerador de números aleatórios RANDOM no pior caso? E no melhor caso? Dê a resposta em termos de notação Q.

7.4 ANÁLISE DO QUICKSORT

A Seção 7.2 nos deu uma ideia do comportamento do pior caso do quicksort e do motivo por que esperamos que ele funcione rapidamente. Nesta seção, analisaremos o comportamento do quicksort mais rigorosamente. Começaremos com uma análise do pior caso, que se aplica a QUICKSORT ou a RANDOMIZED-QUICKSORT, e concluímos com uma análise do tempo de execução esperado de RANDOMIZED-QUICKSORT.

7.4.1 ANÁLISE DO PIOR CASO

Vimos na Seção 7.2 que uma divisão do pior caso em todo nível de recursão do quicksort produz um tempo de execução Q(n_2) que, intuitivamente, é o tempo de execução do pior caso do algoritmo. Agora, vamos provar essa afirmação.

Usando o método de substituição (veja Seção 4.3), podemos mostrar que o tempo de execução do quicksort é $O(n_2)$. Seja $T(n)$ o tempo do pior caso para o procedimento QUICKSORT para uma entrada de tamanho n. Temos a recorrência

$$T(n) \le \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n), \tag{7.1}$$

onde o parâmetro q está na faixa de 0 a $n - 1$ porque o procedimento PARTITION produz dois subproblemas com tamanho total $n - 1$. Nosso palpite é que $T(n) \le cn_2$ para alguma constante c. Substituindo esse palpite na recorrência (7.1), obtemos

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

A expressão $q_2 + (n - q - 1)^2$ atinge um máximo na faixa do parâmetro $0 \le q \le n - 1$ em qualquer das extremidades. Para verificar essa afirmativa, observe que a derivada de segunda ordem da expressão em relação a q é positiva (veja o Exercício 7.4-3). Essa observação nos dá o limite $\max_{0 \le q \le n-1}(q^2 + (n - q - 1)^2) \le (n - 1)^2 = n^2 - 2n + 1$. Continuando com nossa definição do limite de $T(n)$, obtemos

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

visto que podemos escolher a constante c suficientemente grande de modo que o termo $c(2n - 1)$ domine o termo Q(n). Assim, $T(n) = O(n_2)$. Vimos na Seção 7.2 um caso específico no qual o quicksort demora o tempo (n_2): quando o particionamento é desequilibrado. Como alternativa, o Exercício 7.4-1 pede que você mostre que a recorrência (7.1) tem uma solução $T(n) = (n_2)$. Assim, o tempo de execução (do pior caso) do quicksort é Q(n_2).

7.4.2 TEMPO DE EXECUÇÃO ESPERADO

Já vimos a intuição que nos diz por que o tempo de execução do caso médio de RANDOMIZED-QUICKSORT é $O(n \lg n)$: se, em cada nível de recursão, a divisão induzida por RANDOMIZED-PARTITION colocar qualquer fração constante dos elementos em um lado da partição, a árvore de recursão terá a profundidade Q(lg n), e o trabalho $O(n)$ será executado em cada nível. Ainda que acrescentemos alguns novos níveis com a divisão mais desequilibrada possível entre esses níveis, o tempo total permanece $O(n \lg n)$. Podemos analisar precisamente o tempo de execução esperado de RANDOMized-QUICKSORT entendendo, em primeiro lugar, como o procedimento de particionamento funciona, e depois usando essa compreensão para derivar um limite $O(n \lg n)$ para o tempo de execução esperado. Esse limite superior para o tempo de execução esperado, combinado com o limite do melhor caso Q(n lg n) que vimos na Seção 7.2, produz um tempo de execução esperado Q(n lg n). Consideramos, do princípio ao fim, que os valores dos elementos que estão sendo ordenados são distintos.

Tempo de execução e comparações

A única diferença entre os procedimentos QUICKSORT e RANDOMized-QUICKSORT é o modo como selecionam elementos pivôs; em todos os outros aspectos eles são iguais. Portanto, podemos expressar nossa análise de RANDOMIZED-QUICKSORT discutindo os procedimentos QUICKSORT e PARTITION, porém considerando que os elementos pivôs são selecionados aleatoriamente no subarranjo passado para RANDOMIZED-PARTITION.

O tempo de execução do QUICKSORT é dominado pelo tempo gasto no procedimento PARTITION. Toda vez que é chamado, o procedimento PARTITION seleciona um elemento pivô, e esse elemento nunca é incluído em nenhuma chamada recursiva futura a QUICKSORT e PARTITION. Assim, pode haver, no máximo, n chamadas a PARTITION durante a execução inteira do algoritmo de quicksort. Uma chamada a PARTITION demora o tempo $O(1)$ mais uma quantidade de tempo proporcional ao número de iterações do laço **for** nas linhas 3−6. Cada iteração desse laço **for** executa uma comparação na linha 4, comparando o elemento pivô com outro elemento do arranjo A. Portanto, se pudermos contar o número total de vezes que a linha 4 é executada, poderemos limitar o tempo total gasto no laço **for** durante toda a execução de QUICKSORT.

Lema 7.1

Seja X o número de comparações executadas na linha 4 de PARTITION por toda a execução de QUICKSORT para um arranjo de n elementos. Então, o tempo de execução do QUICKSORT é $O(n + X)$.

Prova Pela discussão anterior, o algoritmo faz, no máximo, n chamadas a PARTITION, cada uma das quais faz uma quantidade constante de trabalho e depois executa o laço **for** um certo número de vezes. Cada iteração do laço **for** executa a linha 4.

Portanto, nossa meta é calcular X, o número total de comparações executadas em todas as chamadas a PARTITION. Não tentaremos analisar quantas comparações são feitas em *cada* chamada a PARTITION. Em vez disso, deduziremos um limite global para o número total de comparações. Para tal, temos de entender quando o algoritmo compara dois elementos do arranjo e quando não compara. Para facilitar a análise, renomeamos os elementos do arranjo A como z_1, z_2, ..., z_n, sendo z_i o i-ésimo menor elemento. Também definimos o conjunto $Z_{ij} = \{z_i, z_i + 1, ..., z_j\}$ como o conjunto de elementos entre z_i e z_j, inclusive.

Quando o algoritmo compara z_i e z_j? Para responder a essa pergunta, primeiro observamos que cada par de elementos é comparado no máximo uma vez. Por quê? Os elementos são comparados apenas com o elemento pivô e, depois que uma chamada específica de PARTITION termina, o elemento pivô usado nessa chamada nunca mais é comparado com nenhum outro elemento.

Nossa análise utiliza variáveis aleatórias indicadoras (veja Seção 5.2). Definimos

$$X_{ij} = \mathrm{I}\{z_i \text{ é comparado com } z_j\},$$

onde consideramos a comparação se ocorre em algum instante durante a execução do algoritmo, não apenas durante uma iteração ou uma chamada de PARTITION. Visto que cada par é comparado no máximo uma vez, podemos caracterizar facilmente o número total de comparações executadas pelo algoritmo:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Tomando as esperanças em ambos os lados e depois usando linearidade de esperança e o Lema 5.1, obtemos

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ é comparado com } z_j\}. \qquad (7.2) \end{aligned}$$

resta calcular Pr $\{z_i$ é comparado com $z_j\}$. Nossa análise supõe que o procedimento RANDOMIZED-PARTITION escolhe cada pivô de modo aleatório e independente.

Vamos pensar no caso em que dois itens *não* são comparados. Considere uma entrada para quicksort dos números 1 a 10 (em qualquer ordem) e suponha que o primeiro elemento pivô seja 7. Então, a primeira chamada a PARTITION separa os números em dois conjuntos: {1, 2, 3, 4, 5, 6} e {8, 9, 10}. Ao fazer isso, o elemento pivô 7 é comparado com todos os outros elementos, mas nenhum número do primeiro conjunto (por exemplo, 2) é ou jamais será comparado com qualquer número do segundo conjunto (por exemplo, 9).

Em geral, visto que supomos que os valores dos elementos são distintos, uma vez escolhido um pivô x com $z_i < x < z_j$, sabemos que z_i e z_j não podem ser comparados em nenhum momento subsequente. Se, por outro lado, z_i for escolhido como um pivô antes de qualquer outro item em Z_{ij}, z_i será comparado com cada item em Z_{ij}, exceto ele mesmo. De modo semelhante, se z_j for escolhido como pivô antes de qualquer outro item em Z_{ij}, então z_j será comparado com cada item em Z_{ij}, exceto ele próprio. Em nosso exemplo, os valores 7 e 9 são comparados porque 7 é o primeiro item de $Z_{7,9}$ a ser escolhido como pivô. Em contraste, 2 e 9 nunca serão comparados porque o primeiro elemento pivô escolhido de $Z_{2,9}$ é 7. Assim, z_i e z_j são comparados se e somente se o primeiro elemento a ser escolhido como pivô de Z_{ij} for z_i ou z_j.

Agora, calculamos a probabilidade de esse evento ocorrer. Antes do ponto em que um elemento de Z_{ij} foi escolhido como pivô, todo o conjunto Z_{ij} está reunido na mesma partição. Por conseguinte, qualquer elemento de Z_{ij} tem igual probabilidade de ser o primeiro escolhido como pivô. Como o conjunto Z_{ij} tem $j - i + 1$ elementos, e visto que os pivôs são escolhidos de modo aleatório e independente, a probabilidade de qualquer elemento dado ser o primeiro escolhido como pivô é $1/(j - i + 1)$. Assim, temos

$$\begin{aligned} \Pr\{z_i \text{ é comparado com } z_j\} &= \Pr\{z_i \text{ ou } z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &= \Pr\{z_i \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &\quad + \Pr\{z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \end{aligned}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1}$$

$$= \frac{2}{j-1+1}. \qquad (7.3)$$

A segunda linha decorre porque os dois eventos são mutuamente exclusivos. Combinando as equações (7.2) e (7.3), obtemos

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Podemos avaliar essa soma usando uma troca de variáveis ($k = j - i$) e o limite para a série harmônica na equação (A.7):

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\ &= \sum_{i=1}^{n-1} O(\lg n) \\ &= O(n \lg n) \end{aligned} \qquad (7.4)$$

Assim, concluímos que, usando RANDOMIZED-PARTITION, o tempo de execução esperado de quicksort é $O(n \lg n)$ quando os valores dos elementos são distintos.

Exercícios

7.4-1 Mostre que, na recorrência

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n),$$

$$T(n) = \Omega(n^2).$$

7.4-2 Mostre que o tempo de execução do melhor caso do quicksort é $(n \lg n)$.

7.4-3 Mostre que a expressão $q_2 + (n - q - 1)^2$ atinge um máximo em $q = 0, 1, ..., n - 1$ quando $q = 0$ ou $q = n - 1$.

7.4-4 Mostre que o tempo de execução esperado do procedimento RANDOMized-QUICKSORT é $(n \lg n)$.

7.4-5 Podemos melhorar o tempo de execução do quicksort na prática tirando proveito do tempo de execução rápido da ordenação por inserção quando sua entrada está "quase" ordenada. Ao chamar o quicksort para um subarranjo com menos de k elementos, deixe-o simplesmente retornar sem ordenar o subarranjo. Após o retorno da chamada de alto nível a quicksort, execute a ordenação por inserção para o arranjo inteiro para concluir o processo de ordenação. Mostre que esse algoritmo de ordenação é executado no tempo esperado $O(nk + n \lg(n/k))$. Como k deve ser escolhido, tanto na teoria quanto na prática?

7.4-6 ★

Considere modificar o procedimento PARTITION escolhendo aleatoriamente três elementos do arranjo A e executando a partição em torno de sua mediana (o valor médio dos três elementos). Dê uma aproximação para a probabilidade de obter na pior das hipóteses uma divisão α para $(1 - \alpha)$ em função de α no intervalo $0 < \alpha < 1$.

Problemas

7-1 Correção da partição de Hoare

A versão de PARTITION dada neste capítulo não é o algoritmo de particionamento original. Apresentamos a seguir o algoritmo de partição original, que deve seu nome a C.A.R. Hoare:

```
HOARE-PARTITION(A, p, r)
1  x = A[p]
2  i = p - 1
3  j = r + 1
4  while TRUE
5      repeat
6          j = j - 1
7      until A[j] ≤ x
8      repeat
9          i = i + 1
10     until A[i] ≥ x
11     if i < j
12         trocar A[i] por A[j]
13     else return j
```

a. Demonstre a operação de HOARE-PARTITION sobre o arranjo $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, mostrando os valores do arranjo e os valores auxiliares após cada iteração do laço **for** das linhas 4−13.

As três perguntas seguintes pedem que você apresente um argumento cuidadoso de que o procedimento HOARE-PARTITION é correto. Levando em conta que o subarranjo $A[p \mathinner{..} r]$contém pelo menos dois elementos, prove que:

b. Os índices i e j são tais que nunca acessamos um elemento de A fora do subarranjo $A[p \mathinner{..} r]$.

c. Quando HOARE-PARTITION termina, ele retorna um valor j tal que $p \leq j < r$.

d. Todo elemento de $A[p \mathinner{..} j]$ é menor ou igual a todo elemento de $A[j + 1 \mathinner{..} r]$ quando HOARE-PARTITION termina.

O procedimento PARTITION da Seção 7.1 separa o valor do pivô (originalmente em $A[r]$) das duas partições que ele forma. Por outro lado, o procedimento HOARE-PARTITION sempre insere o valor do pivô (originalmente em $A[p]$) em uma das duas partições $A[p \mathinner{..} j]$ e $A[j + 1 \mathinner{..} r]$. Visto que $p \leq j < r$, essa divisão é sempre não trivial.

e. Reescreva o procedimento QUICKSORT para usar HOARE-PARTITION.

7-2 QUICKSORT *com elementos de valores iguais*

A análise do tempo de execução esperado do QUICKSORT aleatorizado na Seção 7.4.2 supõe que todos os valores dos elementos são distintos. Neste problema, examinamos o que acontece quando não são.

a. Suponha que todos os valores dos elementos sejam iguais. Qual seria o tempo de execução do quicksort aleatorizado nesse caso?

b. O procedimento PARTITION retorna um índice q tal que cada elemento de $A[p. . q - 1]$ é menor ou igual a $A[q]$ e cada elemento de $A[q + 1. . r]$ é maior que $A[q]$. Modifique o procedimento PARTITION para produzir um procedimento PARTITION'(A, p, r), que permuta os elementos de $A[p \; . .r]$ e retorna dois índices q e t, onde $p \leq q \leq t \leq r$, tal que

- todos os elemento de $A[q. . t]$ são iguais,
- cada elemento de $A[p. . q - 1]$ é menor do que $A[q]$, e
- cada elemento de $A[t + 1]$ é maior que $A[q]$.

Como o procedimento PARTITION, o seu procedimento PARTITION' deve demorar o tempo $Q(r - p)$.

c. Modifique o procedimento RANDOMIZED-PARTITION para chamar PARTITION' e denomine o novo procedimento RANDOMIZED-PARTITION'. Então, modifique o procedimento QUICKSORT para produzir um procedimento QUICKSORT'(p, r) que chama RANDOMIZED-PARTITION' e é recursivo somente em partições de elementos que sabemos que não são iguais uns aos outros.

d. Usando QUICKSORT', como você ajustaria a análise na Seção 7.4.2 para evitar a premissa de que todos os elementos são distintos?

7-3 Análise alternativa do QUICKSORT

Uma análise alternativa do tempo de execução de quicksort aleatorizado focaliza o tempo de execução esperado de cada chamada recursiva individual a RANDOMIZED-QUICKSORT, em vez do número de comparações executadas.

a. Demonstre que, dado um arranjo de tamanho n, a probabilidade de qualquer elemento específico ser escolhido como pivô é $1/n$. Use isso para definir variáveis aleatórias indicadoras $X_i = \text{I}$ {o i-ésimo menor elemento é escolhido como pivô}. Qual é $\text{E}[X_i]$?

b. Seja $T(n)$ uma variável aleatória que denota o tempo de execução do quicksort para um arranjo de tamanho n. Demonstre que

$$E[(T)(n)] = E\left[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \Theta(n))\right]. \qquad (7.5)$$

c. Mostre que podemos reescrever a equação (7.5) como

$$E[T(n)] = \frac{2}{n}\sum_{q=2}^{n-1} E[T(q)] + \Theta(n). \tag{7.6}$$

d. Mostre que

$$\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \tag{7.7}$$

(*Sugestão*: Divida o somatório em duas partes, uma para $k = 2, 3, ..., \quad n/2 - 1$ e uma para $\quad k = \quad n/2, ..., n - 1$.)

d. Usando o limite da equação (7.7), mostre que a recorrência na equação (7.6) tem a solução E[$T(n)$] = Q(n lg n). (*Sugestão*: Mostre, por substituição, que E[$T(n)$] ≤ an log $n - bn$ para n suficientemente grande e para alguma constante positiva a.)

7-4 ***Profundidade de pilha para*** **Quicksort**

O algoritmo Quicksort da Seção 7.1 contém duas chamadas recursivas a ele próprio. Após chamar Partition, o Quicksort ordena recursivamente o subarranjo da esquerda e depois ordena recursivamente o subarranjo da direita. A segunda chamada recursiva em Quicksort não é realmente necessária; podemos evitá-la utilizando uma estrutura de controle iterativa. Essa técnica, denominada ***recursão de cauda***, é automaticamente fornecida por bons compiladores. Considere a versão de quicksort a seguir, que simula a recursão de cauda:

```
Tail-Recursive-Quicksort(A, p, r)
1   while p < r
2        // Particionar e ordenar o subarranjo esquerdo.
3        q = Partition(A, p, r)
4        Tail-Recursive-Quicksort(A. p, q – 1)
5        p = q + 1
```

a. Mostre que Tail-Recursive-Quicksort(A, 1, A. *length*) ordena corretamente o arranjo A.

Os compiladores normalmente, executam procedimentos recursivos usando uma ***pilha*** que contém informações pertinentes, inclusive os valores de parâmetros para cada chamada recursiva. As informações para a chamada mais recente estão na parte superior da pilha, e as informações para a chamada inicial encontram-se na parte inferior. Quando um procedimento é invocado, suas informações são ***empurradas*** sobre a pilha; quando ele termina, suas informações são ***extraídas***. Visto que supomos que os parâmetros arranjos são representados por ponteiros, as informações para cada chamada de procedimento na pilha exigem espaço de pilha $O(1)$. A ***profundidade de pilha*** é a quantidade máxima de espaço de pilha usado em qualquer instante durante uma computação.

b. Descreva um cenário no qual a profundidade de pilha de Tail-Recursive-Quicksort é Q(n) para um arranjo de entrada de n elementos.

c. Modifique o código para Tail-Recursive-Quicksort de tal modo que a profundidade de pilha do pior caso seja Q(lg n). Mantenha o tempo de execução esperado $O(n \lg n)$ do algoritmo.

7-5 Partição de mediana de 3

Um modo de melhorar o procedimento RANDOMIZED-QUICKSORT é particionar em torno de um pivô escolhido com maior cuidado que escolher um elemento aleatório do subarranjo. Uma abordagem comum é o método da ***mediana de 3***: escolha como pivô a mediana (o elemento do meio) de um conjunto de 3 elementos selecionados aleatoriamente no subarranjo (veja o Exercício 7.4-6). Para esse problema, vamos supor que os elementos no arranjo de entrada $A[1 .. n]$ sejam distintos e que $n \geq 3$. Denotamos o arranjo de saída ordenado por $A'[1.. n]$. Usando o método da mediana de 3 para escolher o elemento pivô x, defina $p_i = \Pr\{x = A'[i]\}$.

a. Dê uma fórmula exata para p_i em função de n e i para $i = 2, 3, ..., n - 1$. (Observe que $p_1 = p_n = 0$.)

b. De quanto aumentamos a probabilidade de escolher como pivô $x = A'[(n + 1)/2]$, a mediana de $A[1.. n]$, em comparação com a implementação comum? Suponha que $n \to$ e dê o limite da razão dessas probabilidades.

c. Se definirmos que uma "boa" divisão significa escolher o pivô como $x = A'[i]$, onde $n/3 \leq i \leq 2n/3$, de quanto aumentamos a probabilidade de obter uma boa divisão em comparação com a implementação comum? (*Sugestão*: Aproxime a soma por uma integral.)

d. Mostre que, no tempo de execução $(n \lg n)$ do quicksort, o método da mediana de 3 só afeta o fator constante.

7-6 Ordenação nebulosa de intervalos

Considere um problema de ordenação no qual não conhecemos os números exatamente. Em vez disso, para cada número conhecemos um intervalo na linha dos números reais ao qual ele pertence. Isto é, temos n intervalos fechados da forma $[a_i, b_i]$, onde $a_i \leq b_i$. Queremos executar a ***ordenação nebulosa*** desses intervalos, isto é, produzir uma permutação $\langle i_1, i_2, ..., i_n \rangle$ dos intervalos tal que, para $j = 1,2, ..., n$, exista $c_j \in [a_{ij}, b_{ij}]$ que satisfaz $c_1 \leq c_2 \leq ... \leq c_n$.

a. Projete um algoritmo aleatorizado para executar ordenação nebulosa de n intervalos. Seu algoritmo deve ter a estrutura geral de um algoritmo que executa quicksort nas extremidades esquerdas (os valores a_i), mas deve tirar proveito da sobreposição de intervalos para melhorar o tempo de execução. (À medida que os intervalos se sobrepõem mais e mais, o problema da ordenação nebulosa dos intervalos torna-se cada vez mais fácil. Seu algoritmo deve tirar proveito dessa sobreposição até onde ela existir.)

b. Demonstre que seu algoritmo é executado no tempo esperado $Q(n \lg n)$ em geral, mas funciona no tempo esperado $Q(n)$ quando todos os intervalos se sobrepõem (isto é, quando existe um valor x tal que $x \in [a_i, b_i]$ para todo i). O algoritmo não deve verificar esse caso explicitamente; em vez disso, seu desempenho deve melhorar naturalmente à medida que a proporção de sobreposição aumentar.

NOTAS DO CAPÍTULO

O procedimento quicksort foi inventado por Hoare [170]; a versão de Hoare aparece no Problema 7-1. O procedimento PARTITION dado na Seção 7.1 se deve a N. Lomuto. A análise da Seção 7.4 se deve a Avrim Blum. Sedgewick [305] e Bentley [43] nos dão uma boa referência sobre os detalhes de implementação e como eles são importantes.

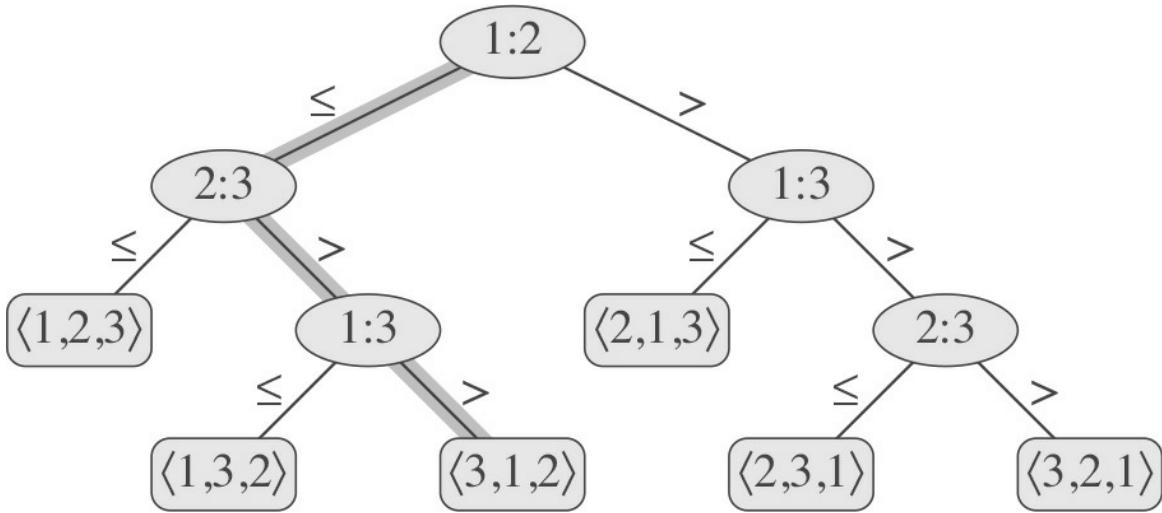
McIlroy [248] mostrou como gerar um "adversário matador" que produz um arranjo para o qual praticamente qualquer implementação do quicksort demora o tempo $Q(n_2)$. Se a implementação for aleatorizada, o adversário produz o arranjo depois de examinar as escolhas aleatórias do algoritmo do quicksort.

8 ORDENAÇÃO EM TEMPO LINEAR

Apresentamos até agora vários algoritmos que podem ordenar n números no tempo $O(n \lg n)$. A ordenação por intercalação e a ordenação por heap atingem esse limite superior no pior caso; o quicksort o atinge na média. Além do mais, para cada um desses algoritmos, podemos produzir uma sequência de n números de entrada que faz o algoritmo ser executado no tempo $(n \lg n)$.

Esses algoritmos compartilham uma propriedade interessante: *a sequência ordenada que eles determinam* se *baseia apenas em comparações entre os elementos da entrada*. Denominamos esses algoritmos de ordenação ***ordenações por comparação***. Todos os algoritmos de ordenação apresentados até aqui são ordenações por comparação.

Na Seção 8.1, provaremos que, para ordenar n elementos, qualquer ordenação por comparação deve efetuar $(n \lg n)$ comparações no pior caso. Assim, a ordenação por intercalação e a ordenação por heap são assintoticamente ótimas e não existe nenhuma ordenação por comparação que seja mais rápida por mais de um fator constante.

As Seções 8.2, 8.3 e 8.4 examinam três algoritmos de ordenação — ordenação por contagem, ordenação digital e ordenação por balde — que são executados em tempo linear. É claro que esses algoritmos utilizam outras operações diferentes de comparações para determinar a sequência ordenada. Por consequência, o limite inferior $(n \lg n)$ não se aplica a eles.

8.1 LIMITES INFERIORES PARA ORDENAÇÃO

Em uma ordenação por comparação, usamos somente comparações entre elementos para obter informações de ordem para uma sequência de entrada $\langle a_1, a_2, ..., a_n \rangle$. Isto é, dados dois elementos a_i e a_j, executamos um dos testes $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i > a_j$ ou $a_i \geq a_j$, para determinar sua ordem relativa. Não podemos inspecionar os valores dos elementos nem obter informações de ordem sobre eles de qualquer outro modo.

Nesta seção, consideramos, sem perder a generalidade, que todos os elementos de entrada são distintos. Adotada essa premissa, comparações da forma $a_i = a_j$ são inúteis, portanto podemos admitir que nenhuma comparação desse tipo é feita. Também observamos que as comparações $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ e $a_i < a_j$ são equivalentes, já que produzem informações idênticas sobre a ordem relativa de a_i e a_j. Portanto, consideramos que todas as comparações têm a forma $a_i \leq a_j$.

O modelo de árvore de decisão

Podemos imaginar as ordenações por comparação como árvores de decisão. Uma ***árvore de decisão*** é uma árvore binária cheia que representa as comparações entre elementos executadas por um determinado algoritmo de ordenação aplicado a uma entrada de dado tamanho. Controle, movimentação de dados e todos os outros aspectos do algoritmo são ignorados. A Figura 8.1 mostra a árvore de decisão correspondente ao algoritmo de ordenação por inserção da Seção 2.1, aplicado a uma sequência de entrada de três elementos.

Figura 8.1 Árvore de decisão para ordenação por inserção para três elementos. Um nó interno anotado como *i*:*j* indica uma comparação entre a_i e a_i. Uma folha anotada como permutação ⟨p(1), p(2), ..., p(*n*)⟩ indica a ordenação $a_{\pi(1)} \le a_{\pi(2)} \le ... \le a_{\pi(n)}$. O caminho sombreado indica as decisões tomadas durante a ordenação da sequência de entrada ⟨$a_1 = 6$, $a_2 = 8$, $a_3 = 5$⟩; a permutação ⟨3, 1, 2⟩ na folha indica que a sequência ordenada é $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. Existem 3! = 6 permutações possíveis dos elementos de entrada; assim, a árvore de decisão deve ter no mínimo seis folhas.

Em uma árvore de decisão, anotamos cada nó interno como *i*:*j* para algum *i* e *j* na faixa $1 \le i, j \le n$, onde *n* é o número de elementos na sequência de entrada. Também anotamos cada folha como uma permutação ⟨*p*(1), *p*(2), ..., *p*(*n*)⟩ (a Seção C.1 dá os fundamentos de permutações). A execução do algoritmo de ordenação corresponde a traçar um caminho simples desde a raiz da árvore de decisão até uma folha. Cada nó interno indica uma comparação $a_i \le a_j$. Então, a subárvore da esquerda determina comparações subsequentes, já que sabemos que $a_i \le a_j$ e a subárvore da direita determinam comparações subsequentes sabendo que $a_i > a_j$. Quando chegamos a uma folha, o algoritmo de ordenação estabeleceu a ordenação $a_{p(1)} \le a_{p(2)} \le ... \le a_{p(n)}$. Como qualquer algoritmo de ordenação correto deve ser capaz de produzir cada permutação de sua entrada, cada uma das *n*! permutações em *n* elementos deve aparecer como uma das folhas da árvore de decisão para uma ordenação por comparação ser correta. Além disso, cada uma dessas folhas deve ser acessível a partir da raiz por um caminho descendente correspondente a uma execução propriamente dita da ordenação por comparação (essas folhas serão denominadas "acessíveis"). Assim, consideraremos apenas árvores de decisão nas quais cada permutação aparece como uma folha acessível.

Um limite inferior para o pior caso

O comprimento do caminho simples mais longo desde a raiz de uma árvore de decisão até qualquer de suas folhas acessíveis representa o número de comparações do pior caso que o algoritmo de ordenação correspondente executa. Consequentemente, o número de comparações do pior caso para dado algoritmo de ordenação por comparação é igual à altura de sua árvore de decisão. Um limite inferior para as alturas de todas as árvores de decisão nas quais cada permutação aparece como uma folha acessível é, portanto, um limite inferior para o tempo de execução de qualquer algoritmo de ordenação por comparação. O teorema a seguir estabelece esse limite inferior.

Teorema 8.1

Qualquer algoritmo de ordenação por comparação exige (*n* lg *n*) comparações no pior caso.

Prova Pela discussão precedente, basta determinar a altura de uma árvore de decisão na qual cada permutação aparece como uma folha acessível. Considere uma árvore de decisão de altura h com l folhas acessíveis correspondente a uma ordenação por comparação sobre n elementos.

Como cada uma das $n!$ permutações da entrada aparece como alguma folha, temos $n! \leq l$. Visto que uma árvore binária de altura h não tem mais de 2^h folhas, temos

$$n! \leq l \leq 2^h,$$

que, tomando logaritmos, implica

$$\begin{aligned} h &\geq \lg(n!) && \text{(já que a função lg é monotonicamente crescente)} \\ &= \Omega(n \lg n) && \text{(pela equação (3.19)).} \end{aligned}$$

Corolário 8.2

A ordenação por heap e a ordenação por intercalação são ordenações por comparação assintoticamente ótimas.

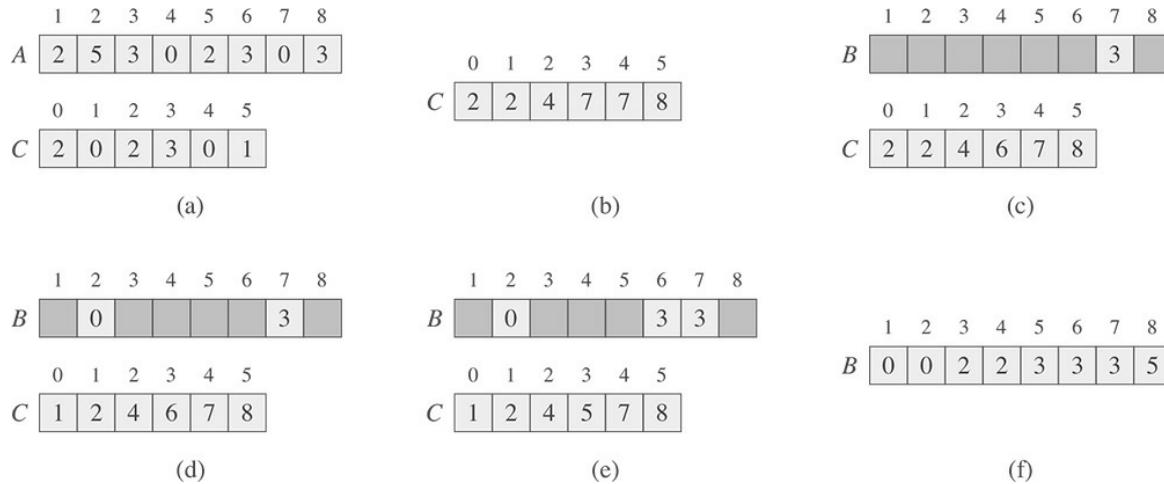
Prova Os limites superiores $O(n \lg n)$ para os tempos de execução para ordenação por heap e ordenação por intercalação correspondem ao limite inferior $(n \lg n)$ do pior caso do Teorema 8.1.

Exercícios

8.1-1 Qual é a menor profundidade possível de uma folha em uma árvore de decisão para ordenação por comparação?

8.1-2 Obtenha limites assintoticamente justos para $\lg(n!)$ sem usar a aproximação de Stirling. Em vez disso, avalie o somatório $\sum_{k=1}^{n} \lg k$, empregando técnicas da Seção A.2.

8.1-3 Mostre que não existe nenhuma ordenação por comparação cujo tempo de execução seja linear para, no mínimo, metade das $n!$ entradas de comprimento n. E no caso de uma fração $1/n$ das entradas de comprimento n? E no caso de uma fração $1/2_n$?

8.1-4 Suponha que você recebeu uma sequência de n elementos para ordenar. A sequência de entrada consiste em n / k subsequências, cada uma contendo k elementos. Os elementos em uma dada subsequência são todos menores que os elementos na subsequência seguinte e maiores que os elementos na subsequência precedente. Assim, para ordenar a sequência inteira de comprimento n, basta ordenar os k elementos em cada uma das n/k subsequências. Mostre um limite inferior $\Omega(n \lg k)$ para o número de comparações necessárias para resolver essa variante do problema de ordenação. (*Sugestão*: Não é rigoroso simplesmente combinar os limites inferiores para as subsequências individuais.)

8.2 ORDENAÇÃO POR CONTAGEM

A ***ordenação por contagem*** supõe que cada um dos n elementos de entrada é um inteiro na faixa 1 a k, para algum inteiro k. Quando $k = O(n)$, a ordenação é executada no tempo $Q(n)$.

A ordenação por contagem determina, para cada elemento de entrada x, o número de elementos menores que x e usa essa informação para inserir o elemento x diretamente em sua posição no arranjo de saída. Por exemplo, se 17

elementos forem menores que x, então x pertence à posição de saída 18. Temos de modificar ligeiramente esse esquema para lidar com a situação na qual vários elementos têm o mesmo valor, já que não queremos inserir todos eles na mesma posição.

No código para ordenação por contagem, consideramos que a entrada é um arranjo $A[1 \,..\, n]$ e, portanto, de $A \cdot comprimento = n$. Precisamos de dois outros arranjos: o arranjo $B[1 \,..\, n]$ contém a saída ordenada e o arranjo $C[0 \,..\, k]$ fornece armazenamento temporário adequado.

Figura 8.2 Operação de Counting-Sort para um arranjo de entrada $A1 \,..\, 8$, onde cada elemento de A é um inteiro não negativo não maior que $k = 5$. (a) O arranjo A e o arranjo auxiliar C após a linha 5. (b) O arranjo C após a linha 8. (c)-(e) O arranjo de saída B e o arranjo auxiliar C após uma, duas e três iterações do laço nas linhas 10-12, respectivamente. Apenas os elementos do arranjo B sombreados em tom mais claro foram preenchidos. (f)) O arranjo de saída ordenado final B.

```
Counting-Sort(A,B,k)
1   seja C[0 .. k] um novo arranjo
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to A·comprimento
5       C[A[j]] = C[A[j]] + 1
6   // C[i] agora contém o número de elementos igual a i.
7   for i = 1 to k
8       C[i] = C[i] + C[i − 1]
9   // C[i] agora contém o número de elementos menores que ou iguais a i.
10  for j = A·comprimento downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

A Figura 8.2 ilustra a ordenação por contagem. Após o laço **for** das linhas 2–3 inicializar o arranjo C para todos os zeros, o laço **for** das linhas 4–5 inspeciona cada elemento da entrada. Se o valor de um elemento de entrada é i, incrementamos $C[i]$. Assim, depois da linha 5, $C[i]$ contém o número de elementos de entrada igual a i para cada inteiro $i = 0, 1, ..., k$. As linhas 7–8 determinam para cada $i = 0, 1, ..., k$ quantos elementos de entrada são menores ou iguais a i, mantendo uma soma atualizada do arranjo C.

Finalmente, o laço **for** das linhas 10–12 coloca cada elemento $A[j]$ em sua posição ordenada correta no arranjo de saída B. Se todos os n elementos forem distintos, quando entrarmos pela primeira vez a linha 10, para cada $A[j]$, o valor $C[A[j$ será a posição final correta de $A[j]$ no arranjo de saída, já que existem $C[A[j$ elementos menores ou iguais a $A[j]$. Como os elementos poderiam não ser distintos, decrementamos $C[A[j$ toda vez que inserimos um valor $A[j]$ no arranjo B. Decrementar $C[A[j$ faz com que o próximo elemento de entrada com valor igual a $A[j]$, se existir algum, vá para a posição imediatamente anterior a $A[j]$ no arranjo de saída.

Quanto tempo a ordenação por contagem exige? O laço **for** das linhas 2–3 demora o tempo Q(k), o laço **for** das linhas 4–5 demora o tempo Q(n), o laço **for** das linhas 7–8 demora o tempo Q(k) e o laço **for** das linhas 10–12 demora o tempo Q(n). Assim, o tempo total é Q($k + n$). Na prática, normalmente usamos a ordenação por contagem quando temos $k = O(n)$, caso em que o tempo de execução é Q(n).

A ordenação por contagem supera o limite inferior de (n lg n) demonstrado na Seção 8.1 porquenãoéumaordenaçãoporcomparação. Defato, nenhumacomparaçãoentreelementos de entrada ocorre em qualquer lugar no código. Em vez disso, a ordenação por contagem utiliza os valores reais dos elementos para efetuar a indexação em um arranjo. O limite inferior (n lg n) para ordenação não se aplica quando nos afastamos do modelo de ordenação por comparação.

Uma propriedade importante da ordenação por contagem é ser ***estável***: números com o mesmo valor aparecem no arranjo de saída na mesma ordem em que aparecem no arranjo de entrada. Isto é, ela desempata dois números segundo a regra de que qualquer número que aparecer primeiro no arranjo de entrada aparecerá primeiro no arranjo de saída. Normalmente, a propriedade de estabilidade só é importante quando dados satélites são transportados juntamente com o elemento que está sendo ordenado. A estabilidade da ordenação por contagem é importante por outra razão: a ordenação por contagem é usada frequentemente como uma sub-rotina em ordenação digital. Como veremos na próxima seção, para que a ordenação digital funcione corretamente, a ordenação por contagem deve ser estável.

Exercícios

8.2-1 Usando a Figura 8.2 como modelo, ilustre a operação de Counting-Sort sobre o arranjo $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2-2 Prove que Counting-Sort é estável.

8.2-3 Suponha que reescrevêssemos o cabeçalho do laço **for** na linha 10 do procedimento Counting-Sort como:

10 **for** $j = 1$ **to** $A \cdot comprimento$

Mostre que o algoritmo ainda funciona corretamente. O algoritmo modificado é estável?

8.2-4 Descreva um algoritmo que, dados n inteiros na faixa 0 a k, reprocesse sua entrada e depois responde a qualquer consulta sobre quantos dos n inteiros caem em uma faixa $[a .. b]$ no tempo $O(1)$. Seu algoritmo deve utilizar o tempo de pré-processamento Q($n + k$).

8.3 Ordenação digital

Ordenação digital é o algoritmo usado pelas máquinas de ordenação de cartões que agora só encontramos em museus de computadores. Os cartões têm 80 colunas, e em cada coluna uma máquina pode fazer uma perfuração em uma das 12 posições. O ordenador pode ser "programado" mecanicamente para examinar determinada coluna de cada cartão em uma pilha e distribuir o cartão em uma das 12 caixas, dependendo do local em que foi perfurado. Então, um operador pode reunir os cartões caixa por caixa, de modo que aqueles que tenham a primeira posição perfurada fiquem sobre os cartões que tenham a segunda posição perfurada, e assim por diante.

No caso de dígitos decimais, cada coluna utiliza apenas 10 posições (as outras duas posições são reservadas para codificar caracteres não numéricos). Então, um número de *d* dígitos ocuparia um campo de *d* colunas. Visto que o ordenador de cartões pode examinar apenas uma coluna por vez, o problema de ordenar *n* cartões em um número de *d* dígitos requer um algoritmo de ordenação.

Intuitivamente, poderíamos ordenar números sobre seu dígito *mais significativo*, ordenar cada uma das caixas resultantes recursivamente e depois combinar as pilhas em ordem. Infelizmente, como os cartões em nove das 10 caixas devem ser postos de lado para ordenar cada uma das caixas, esse procedimento gera muitas pilhas intermediárias de cartões que devem ser controladas (veja o Exercício 8.3-5).

A ordenação digital resolve o problema da ordenação de cartões — contra a intuição normal — ordenando primeiro pelo dígito *menos significativo*. Então, o algoritmo combina os cartões em uma única pilha, sendo que os cartões na caixa 0 precedem os cartões na caixa 1, que precedem os cartões na caixa 2, e assim por diante. Então, ele ordena novamente a pilha inteira pelo segundo dígito menos significativo e recombina a pilha de maneira semelhante. O processo continua até que os cartões tenham sido ordenados por todos os *d* dígitos. Notável é que, nesse ponto, os cartões estão completamente ordenados pelo número de *d* dígitos. Assim, a ordenação exige apenas *d* passagens pela pilha. A Figura 8.3 mostra como a ordenação digital funciona para uma “pilha” de sete números de três dígitos.

Para que a ordenação digital funcione corretamente, as ordenações de dígitos devem ser estáveis. A ordenação executada por um ordenador de cartões é estável, mas o operador tem de tomar cuidado para não trocar a ordem dos cartões à medida que eles saem de uma caixa, ainda que todos os cartões em uma caixa tenham o mesmo dígito na coluna escolhida.

Em um computador típico, que é uma máquina sequencial de acesso aleatório, às vezes, usamos ordenação digital para ordenar registros de informações chaveados por vários campos. Por exemplo, podemos querer ordenar datas por três chaves: ano, mês e dia. Podemos executar um algoritmo de ordenação com uma função de comparação que, dadas duas datas, compare anos e, se houver um empate, compare meses e, se ocorrer outro empate, compare dias. Alternativamente, podemos ordenar as informações três vezes com uma ordenação estável: primeiro pelo dia, em seguida pelo mês e, finalmente, pelo ano.

O código para ordenação digital é direto. O procedimento a seguir considera que cada elemento no arranjo de *n* elementos *A* tem *d* dígitos, onde o dígito 1 é o dígito de ordem mais baixa e o dígito *d* é o dígito de ordem mais alta.

RADIX-SORT (A, d)
1 **for** $i = 1$ **to** d
2 usar uma ordenação estável para ordenar o arranjo A sobre o dígito i

Lema 8.3

Dados *n* números de *d* dígitos nos quais cada dígito pode adotar até *k* valores possíveis, Radix--Sort ordena corretamente esses números no tempo Q(*d*(*n* + *k*)) se a ordenação estável levar o tempo Q(*n* + *k*).

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figura 8.3 Operação de ordenação digital sobre uma lista de sete números de três dígitos. A primeira coluna é a entrada. As colunas restantes mostram a lista após ordenações sucessivas de posições significativas de dígitos em ordem crescente. O sombreamento indica a posição do dígito ordenado para produzir cada lista a partir da anterior.

Prova A correção da ordenação digital decorre por indução para a coluna que está sendo ordenada (veja o Exercício 8.3-3). A análise do tempo de execução depende da ordenação estável usada como algoritmo de ordenação intermediária. Quando cada dígito está na faixa de 0 a $k - 1$ (de modo que pode adotar até k valores possíveis) e k não é muito grande, a ordenação por contagem é a escolha óbvia. Então, cada passagem sobre n números de d dígitos leva o tempo $Q(n + k)$. Há d passagens e, assim, o tempo total para ordenação digital é $Q(d(n + k))$.

Quando d é constante e $k = O(n)$, podemos executar ordenação digital em tempo linear. De modo mais geral, temos alguma flexibilidade quanto aos modos de desmembrar cada chave em dígitos.

Lema 8.4

Dados n números de b bits e qualquer inteiro positivo $r \leq b$, Radix-Sort ordena corretamente esses números no tempo $Q((b/r)(n + 2^r))$ se a ordenação estável que o procedimento usa levar o tempo $Q(n + k)$ para entradas na faixa de 0 a k.

Prova Para um valor $r \leq b$, enxergamos cada chave como composta de $d = b/r$ dígitos de r bits cada. Cada dígito é um inteiro na faixa 0 a $2^r - 1$, de modo que podemos usar a ordenação por contagem com $k = 2^r - 1$. (Por exemplo, podemos considerar que uma palavra de 32 bits tem quatro dígitos de oito bits, de modo que $b = 32$, $r = 8$, $k = 2^r - 1 = 255$ e $d = b/r = 4$.) Cada passagem da ordenação por contagem leva o tempo $Q(n + k) = Q(n + 2^r)$ e há d passagens, o que dá um tempo de execução total de $Q(d(n + 2^r)) = Q((b/r)(n + 2^r))$.

Para valores de n e b dados, desejamos escolher o valor de r, com $r \leq b$, que minimiza a expressão $(b/r)(n + 2^r)$. Se $b < \lg n$, para qualquer valor de $r \leq b$, temos que $(n + 2^r) = Q(n)$. Assim, escolher $r = b$ produz um tempo de execução $(b/b)(n + 2^b) = Q(n)$, que é assintoticamente ótimo. Se $b \geq \lg n$, escolher $r = \lg n$ dá o melhor tempo dentro de um fator constante, que podemos ver da seguinte maneira: escolher $r = \lg n$ produz um tempo de execução $Q(bn/\lg n)$. À medida que aumentamos r acima de $\lg n$, o termo 2^r no numerador aumenta mais rapidamente que o termo r no denominador e, assim, aumentar r acima de $\lg n$ resulta em um tempo de execução $(bn/\lg n)$. Se, em vez disso, diminuirmos r abaixo de $\lg n$, o termo b/r aumentará e o termo $n + 2^r$ permanecerá em $Q(n)$.

A ordenação digital é preferível a um algoritmo de ordenação baseado em comparação, como o quicksort? Se $b = O(\lg n)$, como é frequentemente o caso, e escolhemos $r \approx \lg n$, o tempo de execução de ordenação digital será $Q(n)$,

que parece ser melhor que o tempo de execução esperado do quicksort, Q(n lg n). Porém, os fatores constantes ocultos na notação Q são diferentes. Embora a ordenação digital possa executar menos passagens que o quicksort sobre as n chaves, cada passagem da ordenação digital pode tomar um tempo significativamente maior. Determinar qual algoritmo preferimos depende das características das implementações, da máquina subjacente (por exemplo, muitas vezes, o quicksort utiliza caches de hardware mais eficientemente que a ordenação digital) e dos dados de entrada. Além disso, a versão de ordenação digital que utiliza a ordenação por contagem como ordenação estável intermediária não ordena no lugar, o que muitas das ordenações por comparação de tempo Q(n lg n) fazem. Assim, quando o armazenamento de memória primária é escasso, podemos preferir uma algoritmo de ordenação no lugar como o quicksort.

Exercícios

8.3-1 Usando a Figura 8.3 como modelo, ilustre a operação de Radix-Sort sobre a seguinte lista de palavras em inglês: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

8.3-2 Quais dos seguintes algoritmos de ordenação são estáveis: ordenação por inserção, ordenação por intercalação, ordenação por heap e quicksort? Forneça um esquema simples que torne estável qualquer algoritmo de ordenação. Quanto tempo e espaço adicional seu esquema requer?

8.3-3 Use indução para provar que a ordenação digital funciona. Onde sua prova precisa adotar a premissa de que a ordenação intermediária é estável?

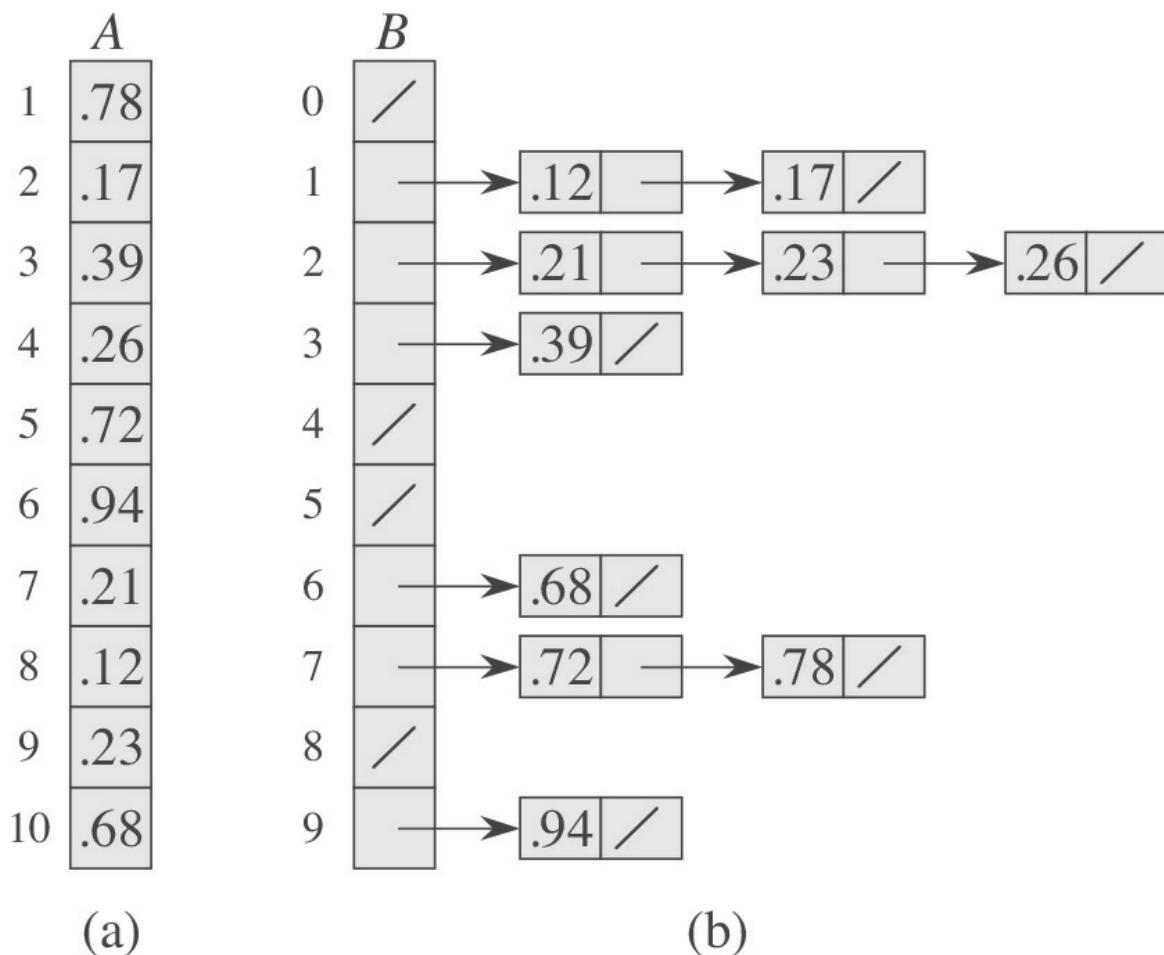
8.3-4 Mostre como ordenar n inteiros na faixa de 0 a $n_3 - 1$ no tempo $O(n)$.

8.3-5 ★ No primeiro algoritmo de ordenação de cartões desta seção, exatamente quantas passagens de ordenação são necessárias para ordenar números decimais de d dígitos no pior caso? Quantas pilhas de cartões um operador precisa controlar no pior caso?

8.4 Ordenação por balde

A ordenação por balde (bucketsort) supõe que a entrada é retirada de uma distribuição uniforme e tem tempo de execução do caso médio de O(n). Como a ordenação por contagem, a ordenação por balde é rápida porque admite alguma coisa em relação à entrada. Enquanto a ordenação por contagem considera que a entrada consiste em inteiros contidos em uma pequena faixa, a ordenação por balde admite que a entrada é gerada por um processo aleatório que distribui elementos de um modo uniforme e independente no intervalo [0, 1) (veja na Seção C.2 uma definição de distribuição uniforme).

A ordenação por balde divide o intervalo [0, 1) em n subintervalos de tamanhos iguais, ou ***baldes***, e depois distribui os n números de entrada entre os baldes. Visto que as entradas são distribuídas de modo uniforme e independente por [0, 1), não esperamos que muitos números caiam em cada balde. Para produzir a saída, simplesmente ordenamos os números em cada balde e depois percorremos os baldes em ordem, anotando os elementos contidos em cada um.

Nosso código para ordenação por balde considera que a entrada é um arranjo de n elementos A e que cada elemento $A[i]$ no arranjo satisfaz $0 \leq A[i] < 1$. O código exige um arranjo auxiliar $B[0 .. n - 1]$ de listas ligadas (baldes) e considera que existe um mecanismo para manter tais listas (a Seção 10.2 descreve como implementar operações básicas para listas ligadas).

BUCKET-SORT(A)
1 seja $B[0 \mathinner{..} n-1]$ um novo arranjo
2 $n = A \cdot comprimento$
3 **for** $i = 0$ **to** $n - 1$
4 faça $B[i]$ uma lista vazia
5 **for** $i = 1$ **to** n
6 insira $A[i]$ na lista $B[\lfloor n\, A[i] \rfloor]$
7 **for** $i = 0$ **to** $n - 1$
8 ordene a lista $B[i]$ com ordenação por inserção
9 concatene as listas $B[0], B[1], \ldots, B[n-1]$ em ordem

A Figura 8.4 mostra a operação de ordenação por balde para um arranjo de entrada de 10 números.

Para ver que esse algoritmo funciona, considere dois elementos $A[i]$ e $A[j]$. Admita, sem perda de generalidade, que $A[i] \le A[j]$. Visto que $nA[i$ $\le nA[j$, o elemento $A[i]$ é inserido no mesmo balde que $A[j]$ ou em um balde com índice mais baixo. Se $A[i]$ e $A[j]$ entrarem no mesmo balde, o laço **for** das linhas 7–8 os colocará na ordem adequada. Se $A[i]$ e $A[j]$ entrarem em baldes diferentes, a linha 9 os colocará na ordem adequada. Portanto, a ordenação por balde funciona corretamente.

Para analisar o tempo de execução, observe que todas as linhas exceto a linha 8 demoram o tempo $O(n)$ no pior caso. Precisamos analisar o tempo total tomado pelas n chamadas à ordenação por inserção na linha 8.

Figura 8.4 Operação de BUCKET-SORT para $n = 10$. (**a**) Arranjo de entrada $A1 \mathinner{..} 10$. (**b**) Arranjo $B0 \mathinner{..} 9$ de listas ordenadas (baldes) após a linha 8 do algoritmo. O balde i contém valores no intervalo semi-aberto $i/10, (i + 1)/10)$. A saída ordenada consiste em uma concatenação ordenada das listas $B0, B1, \ldots, B9$.

Para analisar o custo das chamadas à ordenação por inserção, seja n_i a variável aleatória que denota o número de elementos inseridos no balde $B[i]$. Visto que a ordenação por inserção funciona em tempo quadrático (veja Seção 2.2), o tempo de execução de ordenação por balde é

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2).$$

Analisaremos agora o tempo de execução do caso médio da ordenação por balde calculando o valor esperado do tempo de execução e considerando a esperança para a distribuição da entrada. Considerando esperança de ambos os lados e usando a linearidade de esperança, temos

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] && \text{(por linearidade de esperança)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) && \text{(pela equação (C.22))} \qquad (8.1) \end{aligned}$$

Afirmamos que

$$E[n_i^2] = 2 - 1/n \qquad (8.2)$$

para $i = 0, 1, ..., n - 1$. Não é nenhuma surpresa que cada balde i tenha o mesmo valor de $E[n2\ ]$, já que cada valor no arranjo de entrada A tem igual probabilidade de cair em qualquer balde. Para provar a equação (8.2), definimos variáveis aleatórias indicadoras

$X_{ij} = \mathrm{I}\ \{A[j]$ cai no balde $i\}$

para $i = 0, 1, ..., n - 1$ e $j = 1, 2, ..., n$. Assim,

$$n_i = \sum_{j=i}^{n} X_{ij}.$$

Para calcular $E[n2\ _i]$ expandimos o quadrado e reagrupamos termos:

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\ &= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\ &= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\ &= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le j \\ k\ne j}} E[X_{ij}X_{ik}], \qquad (8.3) \end{aligned}$$

onde a última linha decorre por linearidade de expectativa. Avaliamos os dois somatórios separadamente. A variável aleatória indicadora X_{ij} é 1 com probabilidade $1/n$ e 0 em caso contrário e, portanto,

$$E[X_{ij}^2] = 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right)$$
$$= \frac{1}{n}.$$

Quando $k \neq j$, as variáveis X_{ij} e X_{ik} são independentes e, por conseguinte,

$$E[X_{ij} X_{ik}] = E[X_{ij}]E[X_{ik}]$$
$$= \frac{1}{n} \cdot \frac{1}{n}$$
$$= \frac{1}{n^2}.$$

Substituindo esses dois valores esperados na equação (8.3), obtemos

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \neq j}} \frac{1}{n^2}$$
$$= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}$$
$$= 1 + \frac{n-1}{n}$$
$$= 2 - \frac{1}{n},$$

o que prova a equação (8.2).

Usando esse valor esperado na equação (8.1), concluímos que o tempo do caso médio para ordenação por balde é $Q(n) + n \cdot O(2 - 1/n) = Q(n)$.

Mesmo que a entrada não seja retirada de uma distribuição uniforme, a ordenação por balde ainda pode ser executada em tempo linear. Desde que a entrada tenha a propriedade da soma dos quadrados dos tamanhos de baldes linear no número total de elementos, a equação (8.1) nos diz que a ordenação por balde funcionará em tempo linear.

Exercícios

8.4-1 Usando a Figura 8.4 como modelo, ilustre a operação de Bucket-Sort no arranjo $A = \langle 0{,}79, 0{,}13, 0{,}16, 0{,}64, 0{,}39, 0{,}20, 0{,}89, 0{,}53, 0{,}71, 0{,}42\rangle$.

8.4-2 Explique por que o tempo de execução do pior caso para a ordenação por balde é $Q(n_2)$. Qual é a alteração simples no algoritmo que preserva seu tempo de execução linear do caso médio e torna seu tempo de execução do pior caso igual a $O(n \lg n)$?

8.4-3 Seja X uma variável aleatória igual ao número de caras em dois lançamentos de uma moeda não viciada. Qual é $E[X_2]$? Qual é $E^2[X]$?

8.4-4 ★ Temos n pontos no círculo unitário, $p_i = (x_i, y_i)$, tal que $0 < x_i + y_i \leq 1$ para $i = 1, 2, ..., n$.

Suponha que os pontos estejam uniformemente distribuídos; isto é, a probabilidade de encontrar um ponto em qualquer região do círculo é proporcional à área dessa região. Projete um algoritmo com tempo de execução do caso médio de Q(n) para ordenar os n pontos por suas distâncias $d_i = \sqrt{x^2_i + y^2_i}$ em relação à origem. (*Sugestão*: Projete os tamanhos dos baldes em Bucket-Sort para refletir a distribuição uniforme dos pontos no círculo unitário.)

8.4-5 ★ Uma ***função de distribuição de probabilidade*** $P(x)$ para uma variável aleatória X é definida por $P(x) = \Pr\{X \leq x\}$. Suponha que retiremos uma lista de n variáveis aleatórias $X_1, X_2, ..., X_n$ de uma função de distribuição de probabilidade contínua P que pode ser calculada no tempo $O(1)$. Apresente um algoritmo que ordene esses números em tempo linear do caso médio.

Problemas

8-1 ***Limites inferiores probabilísticos para ordenação por comparação***

Neste problema, provamos um limite inferior ($n \lg n$) para o tempo de execução para qualquer ordenação por comparação determinística ou aleatória de n elementos de entrada distintos. Começamos examinando uma ordenação por comparação determinística A com árvore de decisão T_A. Supomos que toda permutação de entradas de A é igualmente provável.

a. Suponha que cada folha de T_A seja rotulada com a probabilidade de ser atingida dada uma entrada aleatória. Prove que exatamente $n!$ folhas são rotuladas por $1/n!$ e que as restantes são rotuladas por 0.

b. Denotamos por $D(T)$ o comprimento do caminho externo de uma árvore de decisão T, isto é, $D(T)$ é a soma das profundidades de todas as folhas de T. Seja T uma árvore de decisão com $k > 1$ folhas, e sejam RT e LT as subárvores direita e esquerda de T. Mostre que $D(T) = D(LT) + D(RT) + k$.

c. Seja $d(k)$ o valor mínimo de $D(T)$ para todas as árvores de decisão T com $k > 1$ folhas. Mostre que $d(k) = \min_{1 \leq i \leq k-1} \{d(i) + d(k - i) + k\}$. (*Sugestão*: Considere uma árvore de decisão T com k folhas que atinja o mínimo. Seja i_0 o número de folhas em LT e $k - i_0$ o número de folhas em RT.)

d. Prove que, para um dado valor de $k > 1$ e i na faixa $1 \leq i \leq k - 1$, a função $i \lg i + (k - i) \lg(k - i)$ é minimizada em $i = k / 2$. Conclua que $d(k) = (k \lg k)$.

e. Prove que $D(T_A) = (n! \lg(n!))$ e conclua que o tempo do caso médio para ordenar n elementos é ($n \lg n$).

Agora, considere uma ordenação por comparação *aleatorizada* B. Podemos estender o modelo de árvore de decisão para tratar a aleatoriedade incorporando dois tipos de nós: os nós de comparação comuns e os nós de "aleatorização". Um nó de aleatorização modela uma escolha aleatória da forma Random(1, r) feita pelo algoritmo B; o nó tem r filhos, cada um com igual probabilidade de ser escolhido durante uma execução do algoritmo.

f. Mostre que, para qualquer ordenação por comparação aleatorizada B, existe uma ordenação por comparação determinística A cujo número de comparações não é maior do que as feitas por B.

8-2 Ordenação no lugar em tempo linear

Suponha que temos um arranjo de n registros de dados para ordenar e que a chave de cada registro tem o valor 0 ou 1. Um algoritmo para ordenar tal conjunto de registros poderia ter algum subconjunto das três características desejáveis a seguir:

1. O algoritmo é executado no tempo $O(n)$.

2. O algoritmo é estável.

3. O algoritmo ordena no lugar utilizando não mais que uma quantidade constante de espaço de armazenamento além do arranjo original.

a. Dê um algoritmo que satisfaça os critérios 1 e 2.

b. Dê um algoritmo que satisfaça os critérios 1 e 3.

c. Dê um algoritmo que satisfaça os critérios 2 e 3.

d. Você pode usar qualquer de seus algoritmos de ordenação dados nas partes (a)-(c) como o método de ordenação utilizado na linha 2 de Radix-Sort de modo que Radix-Sort ordene n registros com chaves de b bits no tempo $O(bn)$? Justifique sua resposta.

e. Suponha que os n registros tenham chaves na faixa de 1 a k. Mostre como modificar a ordenação por contagem de modo que ela ordene os registros no lugar no tempo $O(n + k)$. Você pode usar armazenamento $O(k)$ fora do arranjo de entrada. Seu algoritmo é estável? (*Sugestão*: Como você faria isso para $k = 3$?)

8-3 Ordenação de itens de comprimento variável

a. Você tem um arranjo de inteiros no qual inteiros diferentes podem ter números de dígitos diferentes, mas o número total de dígitos para *todos* os inteiros no arranjo é n. Mostre como ordenar o arranjo no tempo $O(n)$.

b. Você tem um arranjo de cadeias no qual cadeias diferentes podem ter números de caracteres diferentes, mas o número total de caracteres para todas as cadeias é n. Mostre como ordenar as cadeias no tempo $O(n)$.

(Observe que a ordem desejada aqui é a ordem alfabética padrão; por exemplo, $a < ab < b$.)

8-4 Jarros de água

Suponha que você tem n jarros de água vermelhos e n jarros azuis, todos de formas e tamanhos diferentes. Todos os jarros vermelhos contêm quantidades diferentes de água, assim como os jarros azuis. Além disso, para todo jarro vermelho existe um jarro azul que contém a mesma quantidade de água e vice-versa.

Sua tarefa é determinar um agrupamento dos jarros em pares de jarros vermelhos e azuis que contenham a mesma quantidade de água. Para tal, você pode executar a seguinte operação: escolha um par de jarros formado por um jarro vermelho e um jarro azul, encha o jarro vermelho com água e depois despeje a água no jarro azul. Essa operação informará se o jarro vermelho ou o jarro azul pode conter mais água ou que eles têm o mesmo volume. Considere que tal comparação demora uma unidade de tempo. Seu objetivo é encontrar um

algoritmo que faça um número mínimo de comparações para determinar o agrupamento. Lembre-se de que você não pode comparar diretamente dois jarros vermelhos ou dois jarros azuis.

a. Descreva um algoritmo determinístico que use Q(n2) comparações para agrupar os jarros em pares.

b. Prove um limite inferior de (n lg n) para o número de comparações que deve efetuar um algoritmo que resolve esse problema

d. Dê um algoritmo aleatorizado cujo número esperado de comparações é $O(n \lg n)$ e prove que esse limite é correto. Qual é o número de comparações do pior caso de seu algoritmo?

8-5 Ordenação por média

Suponha que, em vez de ordenar um arranjo, queremos simplesmente que os elementos aumentem na média. Mais exatamente, dizemos que o arranjo de n elementos A é ***k-ordenado*** se, para todo i = 1, 2, ..., $n - k$, a seguinte desigualdade é válida:

$$\frac{\sum_{j=1}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=1+1}^{i+k} A[j]}{k}.$$

a. O que significa um arranjo ser 1-ordenado?

b. Dê uma permutação dos números 1, 2, ..., 10 que seja 2-ordenada, mas não ordenada.

c. Prove que um arranjo de n elementos é k-ordenado se e somente se $Ai \leq Ai + k$ para todo i = 1, 2, ..., $n - k$.

d. Dê um algoritmo que execute uma k-ordenação de um arranjo de n elementos no tempo $O(n \lg(n/k))$.

Também podemos mostrar um limite inferior para o tempo para produzir um arranjo k-ordenado quando k é uma constante.

e. Mostre que podemos ordenar um arranjo k-ordenado de comprimento n no tempo $O(n \lg k)$. (*Sugestão*: Use a solução do Exercício 6.5-9.)

f. Mostre que, quando k é uma constante, executar uma k-ordenação em um arranjo de n elementos requer o tempo (n lg n). (*Sugestão*: Use a solução para a parte *e* juntamente com o limite inferior para ordenações por comparação.)

8-6 Limite inferior para a intercalação de listas ordenadas

O problema de intercalar duas listas ordenadas surge com frequência. Vimos um procedimento para tal problema na forma da sub-rotina de Merge na Seção 2.31. Neste problema, mostraremos que existe um limite inferior $2n - 1$ para o número de comparações do pior caso exigidas para intercalar duas listas ordenadas, cada uma contendo n itens. Primeiro, mostraremos um limite inferior de $2n - o(n)$ comparações usando uma árvore de decisão.

a. Dados $2n$ números, calcule o número de modos possíveis de dividi-los em duas listas ordenadas, cada uma com n números.

b. Usando uma árvore de decisão e sua resposta à parte (a), mostre que qualquer algoritmo que intercale corretamente duas listas ordenadas deve executar pelo menos $2n - o(n)$ comparações.

Agora, mostraremos um limite ligeiramente mais restrito $2n - 1$.

c. Mostre que, se dois elementos são consecutivos na sequência ordenada e vêm de listas diferentes, eles devem ser comparados.

d. Use sua resposta à parte anterior para mostrar um limite inferior de $2n$ - 1 comparações para intercalar duas listas ordenadas.

8-7 O lema de ordenação 0-1 e ordenação por coluna

Uma operação de ***comparação e troca*** para dois arranjos de elementos $A[i]$ e $A[j]$, onde i < j, tem a forma

```
COMPARE-EXCHANGE (A, i, j)
1  if A[i] > A[j]
2        exchange A[i] with A[j]
```

Após a operação de comparação e troca, sabemos que $A[i]$ $A[j]$.

Um ***algoritmo de comparação e troca inconsciente*** funciona exclusivamente por uma sequência de operações pré-especificadas de comparação e troca. Os índices das posições comparadas na sequência devem ser determinados com antecedência e, embora possam depender do número de elementos que estão sendo ordenados, não podem depender dos valores que estão sendo ordenados nem do resultado de qualquer operação anterior de comparação e troca. Por exemplo, apresentamos a seguir, a ordenação por inserção expressa como um algoritmo de comparação e troca inconsciente:

```
INSERTION-SORT(A)
1  for j = 2 to A·comprimento
2        for i = j − 1 downto 1
3              COMPARE-EXCHANGE(A, i, i + 1)
```

O ***lema de ordenação 0-1*** nos dá um modo poderoso de provar que um algoritmo de comparação e troca inconsciente produz um resultado ordenado. O lema declara que, se um algoritmo de comparação e troca inconsciente ordenar corretamente todas as sequências de entrada compostas somente por 0s e 1s, então ordenará corretamente todas as entradas que contenham valores arbitrários.

Você provará o lema de ordenação 0-1 provando seu contraposto: se um algoritmo de comparação e troca inconsciente não ordenar uma entrada que contenha valores arbitrários, então não ordenará alguma entrada 0-1. Suponha que um algoritmo de comparação e troca inconsciente X não ordena corretamente o arranjo $A[1 \dots n]$. Seja $A[p]$ o menor valor em A que o algoritmo X coloca na posição errada e seja $A[q]$ o valor que o algoritmo X desloca até a posição para a qual $A[p]$ deveria ter ido. Defina um arranjo $B[1 .. n]$ de 0s e 1s da seguinte maneira:

$$B[i] = \begin{cases} 0 \ se \ A[i] \leq A[p], \\ 1 \ se \ A[i] > A[p]. \end{cases}$$

a. Demonstre que $Aq > Ap$, de modo que $Bp = 0$ e $Bq = 1$.

b. Para concluir a prova do lema de ordenação 0-1, prove que o algoritmo X não ordena o arranjo B corretamente.

Agora usaremos o lema de ordenação 0-1 para provar que um determinado algoritmo de ordenação funciona corretamente. O algoritmo ***de ordenação por coluna (columnsort)*** funciona sobre um arranjo retangular de n elementos. O arranjo tem r linhas e s colunas (de modo que $n = rs$) e está sujeito a três restrições:

- r deve ser par,
- s deve ser um divisor de r e
- $r \geq 2s_2$.

Quando o algoritmo de ordenação por coluna termina, o arranjo está ordenado por **ordem de coluna:** lendo-se as colunas de cima para baixo e da esquerda para a direita, os elementos crescem monotonicamente.

A ordenação por coluna funciona em oito etapas, independentemente do valor de n. As etapas ímpares são todas iguais: ordenam cada coluna individualmente. Cada etapa par é uma permutação fixa. As etapas são:

1. Ordenar cada coluna.

2. Transpor o arranjo e modelar novamente o arranjo em r linhas e s colunas. Em outras palavras, transformar a coluna da extrema esquerda nas r/s linhas superiores, em ordem, transformar a coluna seguinte nas r/s linhas seguintes em ordem, e assim por diante.

3. Ordenar cada coluna.

4. Executar o inverso da permutação efetuada na etapa 2.

5. Ordenar cada coluna.

6. Deslocar a metade superior de cada coluna para a metade inferior da mesma coluna e deslocar a metade inferior de cada coluna para a metade superior da próxima coluna à direita. Deixar vazia a metade superior da coluna da extrema esquerda. Deslocar a metade inferior da última coluna para a metade superior de uma nova coluna na extrema direita e deixar vazia a metade inferior dessa nova coluna.

7. Ordenar cada coluna.

8. Executar o inverso da permutação efetuada na etapa 6.

A Figura 8.5 mostra um exemplo das etapas da ordenação por coluna com $r = 6$ e $s = 3$. (Ainda que viole o requisito de $r \geq 2s_2$, esse exemplo funciona.)

c. Demonstre que podemos tratar a ordenação por coluna como um algoritmo de comparação e troca inconsciente, mesmo que não saibamos qual método de ordenação as etapas ímpares utilizam.

Embora possa parecer difícil acreditar que a ordenação por coluna realmente funciona, você usará o lema da ordenação 0-1 para provar que isso ocorre. O lema de ordenação 0-1 se aplica porque podemos tratar a ordenação por coluna como um algoritmo de comparação e troca inconsciente. Algumas definições o ajudarão a aplicar o lema de ordenação 0-1. Dizemos que uma área de um arranjo é ***limpa*** se soubermos que ela contém só 0s ou só 1s. Caso contrário, a área poderá conter uma mistura de 0s e 1s e é **suja.** Daqui em diante, considere que o arranjo de entrada contém somente 0s e 1s, e que podemos tratá-lo como um arranjo com r linhas e s colunas.

	(a)			(b)			(c)			(d)			(e)	
10	14	5	4	1	2	4	8	10	1	3	6	1	4	11
8	7	17	8	3	5	12	16	18	2	5	7	3	8	14
12	1	6	10	7	6	1	3	7	4	8	10	6	10	17
16	9	11	12	9	11	9	14	15	9	13	15	2	9	12
4	15	2	16	14	13	2	5	6	11	14	17	5	13	16
18	3	13	18	15	17	11	13	17	12	16	18	7	15	18

	(f)				(g)				(h)				(i)	
1	4	11		5	10	16		4	10	16		1	7	13
2	8	12		6	13	17		5	11	17		2	8	14
3	9	14		7	15	18		6	12	18		3	9	15
5	10	16	1	4	11		1	7	13			4	10	16
6	13	17	2	8	12		2	8	14			5	11	17
7	15	18	3	9	14		3	9	15			6	12	18

Figura 8.5 Etapas da ordenação por coluna. **(a)** O arranjo de entrada com 6 linhas e 3 colunas. **(b)** Após ordenar cada coluna na etapa 1. **(c)** Após transpor e remodelar na etapa 2. **(d)** Após ordenar cada coluna na etapa 3. **(e)** Após executar a etapa 4, que inverte a permutação da etapa 2. **(f)** Após ordenar cada coluna na etapa 5. **(g)** Após deslocar meia coluna na etapa 6. **(h)** Após ordenar cada coluna na etapa 7. **(i)** Após executar a etapa 8, que inverte a permutação da etapa 6. Agora o arranjo está ordenado por coluna.

d. Prove que, após as etapas 1-3, o arranjo consiste em algumas linhas limpas de 0s na parte superior, algumas linhas de 1s na parte inferior e, no máximo, s linhas sujas entre elas.

e. Prove que, após a etapa 4, o arranjo, lido em ordem orientada por coluna, começa com uma área limpa de 0s, termina com uma área limpa de 1s e tem uma área suja de, no máximo, s^2 elementos no meio.

f. Prove que as etapas 5-8 produzem uma saída 0-1 totalmente ordenada. Conclua que a ordenação por coluna ordena corretamente todas as entradas que contêm valores arbitrários.

g. Agora suponha que s não seja divisor de r. Prove que, após as etapas 1 - 3, o arranjo consiste em algumas linhas limpas de 0s na parte superior, algumas linhas limpas de 1s na parte inferior e, no máximo, 2s – 1 linhas sujas entre elas. Qual deverá ser o tamanho de r, em comparação com s, para que a ordenação por coluna ordene corretamente quando s não for um divisor de r?

h. Sugira uma mudança simples na etapa 1 que nos permita manter o requisito de $r \geq 2s^2$ mesmo quando s não é um divisor de r, e prove que, com tal mudança, a ordenação por coluna ordena corretamente.

Notas do capítulo

O modelo de árvore de decisão para o estudo de ordenações por comparação foi introduzido por Ford e Johnson [110]. O tratado abrangente de Knuth sobre a ordenação [211] aborda muitas variações do problema da ordenação, inclusive o limite inferior da Teoria da Informação sobre a complexidade da ordenação que demos neste livro. Ben-Or [39] estudou limites inferiores para ordenação utilizando generalizações do modelo de árvore de decisão.

Knuth credita a H. H. Seward a criação da ordenação por contagem, em 1954, e também a ideia de combinar a ordenação por contagem com a ordenação digital. A ordenação digital que começa pelo dígito menos significativo parece ser um algoritmo popular amplamente utilizado por operadores de máquinas mecânicas de ordenação de

cartões. De acordo com Knuth, a primeira referência ao método publicada é um documento de 1929 escrito por L. J. Comrie que descreve o equipamento de perfuração de cartões. A ordenação por balde está em uso desde 1956, quando a ideia básica foi proposta por E. J. Isaac e R. C. Singleton [188].

Munro e Raman [263] apresentam um algoritmo de ordenação estável que executa $O(n_1+)$ comparações no pior caso, onde $0 < \ \leq 1$ é qualquer constante fixa. Embora qualquer dos algoritmos de tempo $O(n \lg n)$ efetue um número menor de comparações, o algoritmo de Munro e Raman move os dados apenas $O(n)$ vezes e opera no lugar.

O caso da ordenação de n inteiros de b bits no tempo $o(n \lg n)$ foi considerado por muitos pesquisadores. Vários resultados positivos foram obtidos, cada um sob premissas um pouco diferentes sobre o modelo de computação e as restrições impostas ao algoritmo. Todos os resultados supuseram que a memória do computador está dividida em palavras endereçáveis de b bits. Fredman e Willard [115] introduziram a estrutura de dados de árvores de fusão e a empregaram para ordenar n inteiros no tempo $O(n \lg n/\lg \lg n)$. Esse limite foi aperfeiçoado mais tarde para o tempo $O(n\sqrt{\lg n})$ por Andersson [16]. Esses algoritmos exigem o uso de multiplicação e de várias constantes pré-calculadas. Andersson, Hagerup, Nilsson e Raman [17] mostraram como ordenar n inteiros no tempo $O(n \lg \lg n)$ sem usar multiplicação, mas seu método exige espaço de armazenamento que pode ser ilimitado em termos de n. Utilizando hashing multiplicativo, podemos reduzir o espaço de armazenamento necessário para $O(n)$, mas então o limite $O(n \lg \lg n)$ do pior caso para o tempo de execução se torna um limite de tempo esperado. Generalizando as árvores de busca exponencial de Andersson [16], Thorup [335] apresentou um algoritmo de ordenação de tempo $O(n(\lg \lg n)^2)$ que não usa multiplicação ou aleatorização e utiliza espaço linear. Combinando essas técnicas com algumas ideias novas, Han [158] melhorou o limite para ordenação até o tempo $O(n \lg \lg n \lg \lg \lg n)$. Embora esses algoritmos sejam inovações teóricas importantes, todos eles são razoavelmente complicados e, até o presente momento, parece improvável que venham a competir na prática com algoritmos de ordenação existentes.

O algoritmo de ordenação por coluna no Problema 8-7 é de Leighton [227].

9 MEDIANAS E ESTATÍSTICAS DE ORDEM

A i-ésima ***estatística de ordem*** de um conjunto de n elementos é o i-ésimo menor elemento. Por exemplo, o ***mínimo*** de um conjunto de elementos é a primeira estatística de ordem ($i = 1$), e o ***máximo*** é a n-ésima estatística de ordem ($i = n$). Informalmente, uma ***mediana*** é o "ponto do meio" do conjunto. Quando n é ímpar, a mediana é única e ocorre em $i = (n + 1)/2$. Quando n é par, existem duas medianas, que ocorrem em $i = n / 2$ e $i = n / 2 + 1$. Assim, independentemente da paridade de n, as medianas ocorrem em $i = (n + 1)/2$ (a ***mediana inferior***) e $i = (n + 1)/2$ (a ***mediana superior***). Todavia, por simplicidade, neste texto usaremos sempre a expressão "mediana" para nos referirmos à mediana inferior.

Este capítulo aborda o problema de selecionar a i-ésima estatística de ordem de um conjunto de n números distintos. Por conveniência, supomos que o conjunto contém números distintos, embora praticamente tudo que fizermos se estenda à situação na qual um conjunto contém valores repetidos. Especificamos o ***problema de seleção*** formalmente do seguinte modo:

Entrada: Um conjunto A de n números (distintos) e um inteiro i, com $1 \leq i \leq n$.

Saída: O elemento $x \in A$, que é maior que exatamente $i - 1$ outros elementos de A. Podemos resolver o problema de seleção no tempo $O(n \lg n)$, já que podemos ordenar os números usando ordenação por heap ou por intercalação e, então, simplesmente indexar o i-ésimo elemento no arranjo de saída. Contudo, existem algoritmos mais rápidos.

Na Seção 9.1, examinamos o problema de selecionar o mínimo e o máximo de um conjunto de elementos. Mais interessante é o problema de seleção geral, que investigamos nas duas seções subsequentes. A Seção 9.2 analisa um algoritmo aleatorizado prático que alcança um tempo de execução esperado $O(n)$ considerando elementos distintos. A Seção 9.3 contém um algoritmo de interesse mais teórico, que alcança o tempo de execução $O(n)$ no pior caso.

9.1 MÍNIMO E MÁXIMO

Quantas comparações são necessárias para determinar o mínimo de um conjunto de n elementos? Podemos obter facilmente um limite superior de $n - 1$ comparações: examine cada elemento do conjunto por vez e mantenha o controle do menor elemento visto até então. No procedimento a seguir, consideramos que o conjunto reside no arranjo A, onde $A \cdot comprimento = n$.

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A·comprimento
3     if min > A[i]
4         min = A[i]
5  return min
```

É claro que também podemos determinar o máximo com $n - 1$ comparações.

Isso é o melhor que podemos fazer? Sim, desde que possamos obter um limite inferior de $n - 1$ comparações para o problema de determinar o mínimo. Imagine qualquer algoritmo que determine o mínimo como um torneio entre os elementos. Cada comparação é uma partida no torneio, na qual o menor dos dois elementos vence. Observando que todo elemento, exceto o vencedor, deve perder pelo menos uma partida, concluímos que são necessárias $n - 1$ comparações para determinar o mínimo. Por consequência, o algoritmo MINIMUM é ótimo com relação ao número de comparações executadas.

Mínimo e máximo simultâneos

Em algumas aplicações, devemos determinar o mínimo e também o máximo de um conjunto de n elementos. Por exemplo, um programa gráfico talvez tenha de ajustar a escala de um conjunto de (x, y) dados para enquadrá-lo em uma tela de exibição retangular ou em outro dispositivo de saída gráfica. Para tal, o programa deve primeiro determinar os valores mínimo e máximo de cada coordenada.

Nesta altura, o procedimento para determinar o mínimo e o máximo de n elementos usando Q(n) comparações, que é o número assintoticamente ótimo, deve ser óbvio: simplesmente determine o mínimo e o máximo independentemente, usando $n - 1$ comparações para cada um deles, o que dá um total de $2n - 2$ comparações.

De fato, podemos determinar ambos, mínimo e máximo, usando no máximo 3 $n/2$ comparações. Para tal, mantemos os elementos mínimo e máximo que vimos até o momento em questão. Em vez de processar cada elemento da entrada comparando-o com o mínimo e o máximo atuais, ao custo de duas comparações por elemento, processamos elementos aos pares. Comparamos pares de elementos da entrada primeiro *uns com os outros* e, depois, comparamos o menor com o mínimo atual e o maior com o máximo atual, ao custo de três comparações para cada dois elementos.

A definição de valores iniciais para o mínimo e o máximo atuais depende de n ser ímpar ou par. Se n é ímpar, igualamos o mínimo e o máximo ao valor do primeiro elemento e processamos os elementos restantes aos pares. Se n é par, executamos uma comparação sobre os dois primeiros elementos para determinar os valores iniciais do mínimo e do máximo, e processamos os elementos restantes aos pares, como no caso de n ímpar.

Vamos analisar o número total de comparações. Se n é ímpar, executamos 3 $n/2$ comparações. Se n é par, executamos uma comparação inicial seguida por $3(n - 2)/2$ comparações, dando um total de $3n/2 - 2$. Assim, em qualquer caso, o número total de comparações é, no máximo, 3 $n/2$.

Exercícios

9.1-1 Mostre que o segundo menor entre n elementos pode ser determinado com $n + \lg n - 2$ comparações no pior caso. (*Sugestão*: Determine também o menor elemento.)

9.1-2 ★

Mostre que, no pior caso, $3n/2 - 2$ comparações é o limite inferior para determinar o máximo e o mínimo entre n números. (*Sugestão*: Considere quantos números são potencialmente o máximo ou o mínimo e investigue como uma comparação afeta essas contagens.)

9.2 SELEÇÃO EM TEMPO LINEAR ESPERADO

O problema de seleção geral parece mais difícil que o problema simples de determinar um mínimo, ainda que surpreendentemente o tempo de execução assintótico para ambos os problemas seja o mesmo: $Q(n)$. Nesta seção, apresentamos um algoritmo de divisão e conquista para o problema de seleção. O algoritmo RANDOMIZED-SELECT é modelado conforme o algoritmo quicksort do Capítulo 7. Como no quicksort, particionamos o arranjo de entrada recursivamente. Porém, ao contrário do quicksort, que processa recursivamente ambos os lados da partição, RANDOMIZED-SELECT funciona somente de um lado da partição. Essa diferença se destaca na análise: enquanto o quicksort tem um tempo de execução esperado $Q(n \lg n)$, o tempo de execução esperado de RANDOMIZED-SELECT é $Q(n)$, supondo que os elementos são distintos.

O RANDOMIZED-SELECT utiliza o procedimento RANDOMIZED-PARTITION introduzido na Seção 7.3. Portanto, como o RANDOMIZED-QUICKSORT, ele é um algoritmo aleatorizado, já que seu comportamento é determinado em parte pela saída de um gerador de números aleatórios. O código para RANDOMIZED-SELECT apresentado a seguir retorna o i-ésimo menor elemento do arranjo $A[p \mathinner{..} r]$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k          // O valor pivô é a resposta
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

O procedimento RANDOMIZED-SELECT funciona da seguinte maneira: a linha 1 verifica se é um caso-base da recursão, no qual o subarranjo $A[p \mathinner{..} r]$ consiste em apenas um elemento. Nesse caso, i deve ser igual a 1, e simplesmente retornamos $A[p]$ na linha 2 como o i-ésimo menor elemento. Caso contrário, a chamada a RANDOMIZED-PARTITION na linha 3 particiona o arranjo $A[p \mathinner{..} r]$ em dois subarranjos $A[p \mathinner{..} q-1]$ e $A[q+1 \mathinner{..} r]$ (possivelmente vazios) tais que cada elemento de $A[p \mathinner{..} q-1]$ é menor ou igual a $A[q]$, que, por sua vez, é menor que cada elemento de $A[q+1 \mathinner{..} r]$. Como no quicksort, nos referiremos a $A[q]$ como o elemento ***pivô***. A linha 4 calcula o número k de elementos no subarranjo $A[p \mathinner{..} q]$, isto é, o número de elementos no lado baixo da partição, mais um para o elemento pivô. Então, a linha 5 verifica se $A[q]$ é o i-ésimo menor elemento. Se for, a linha 6 retorna $A[q]$. Caso contrário, o algoritmo determina em qual dos dois subarranjos $A[p \mathinner{..} q-1]$ e $A[q+1 \mathinner{..} r]$ se encontra o i-ésimo menor elemento. Se $i < k$, o elemento desejado se encontra no lado baixo da partição, e a linha 8 o seleciona recursivamente no subarranjo. Porém, se $i > k$, o elemento desejado se encontra no lado alto da partição. Como já conhecemos k valores que são menores que o i-ésimo menor elemento de $A[p \mathinner{..} r]$ — isto é, os elementos de $A[p \mathinner{..} q]$ —, o elemento desejado é o $(i-k)$-ésimo menor elemento de $A[q+1 \mathinner{..} r]$, que a linha 9 determina recursivamente. O código parece permitir chamadas recursivas a subarranjos com 0 elementos, mas o Exercício 9.2-1 pede que você mostre que essa situação não pode acontecer.

O tempo de execução do pior caso para RANDOMIZED-SELECT é $Q(n_2)$, até mesmo para determinar o mínimo, porque poderíamos ter o grande azar de sempre executar a partição em torno do maior elemento restante e o particionamento levar o tempo $Q(n)$. Entretanto, veremos que o algoritmo tem um tempo de execução esperado linear e, como ele é aleatorizado, nenhuma entrada específica provoca o comportamento do pior caso.

Para analisar o tempo de execução esperado de RANDOMIZED-SELECT, supomos que o tempo de execução de um arranjo de entrada $A[p \mathinner{..} r]$ de n elementos é uma variável aleatória que denotamos por $T(n)$, e obtemos um limite

superior para $E[T(n)]$ da maneira descrita a seguir. O procedimento RANDOMIZED-PARTITION tem igual probabilidade de retornar qualquer elemento como pivô. Portanto, para cada k tal que $1 \le k \le n$, o subarranjo $A[p \,..\, q]$ tem k elementos (todos menores ou iguais ao pivô) com probabilidade $1/n$. Para $k = 1, 2, ..., n$, definimos variáveis aleatórias indicadoras X_k, onde

$X_k = \mathrm{I}$ {o subarranjo $A[p \,..\, q]$ tem exatamente k elementos} ,

e, assim, considerando que os elementos são distintos, temos

$$\mathrm{E}[X_k] = 1/n \,. \tag{9.1}$$

Quando chamamos RANDOMIZED-SELECT e escolhemos $A[q]$ como o elemento pivô, não sabemos *a priori* se terminaremos imediatamente com a resposta correta, se faremos a recursão no subarranjo $A[p \,..\, q-1]$ ou subarranjo $A[q+1 \,..\, r]$. Essa decisão depende de onde o i-ésimo menor elemento cai em relação a $A[q]$. Considerando que $T(n)$ é monotonicamente crescente, podemos impor um limite superior ao tempo necessário para a chamada recursiva pelo tempo necessário para a chamada recursiva para a maior entrada possível. Em outras palavras, para obter um limite superior supomos que o i-ésimo elemento está sempre no lado da partição que tem o maior número de elementos. Para uma dada chamada de RANDOMIZED-SELECT, a variável aleatória indicadora X_k tem o valor 1 para exatamente um valor de k, e é 0 para todos os outros k. Quando $X_k = 1$, os dois subarranjos nos quais poderíamos executar a recursão têm tamanhos $k - 1$ e $n - k$. Consequentemente, temos a recorrência

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n). \end{aligned}$$

Tomando valores esperados, temos

$$\begin{aligned} &E[T(n)] \\ &\le E\left[\sum_{k=1}^{n} X_k \cdot T(\max\,(k-1, n-k)) + O(n)\right] \\ &= \sum_{k=1}^{n} E[X_k \cdot T(\max(k-1, n-k))] + O\,(n) \quad \text{(por linearidade de esperança)} \\ &= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n) \,\text{(pela equação(C.24))} \\ &= \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n) \qquad \text{(pela equação(9.1))}. \end{aligned}$$

Para aplicar a equação (C.24), contamos com o fato que X_k e $T(\max(k-1, n-k))$ são variáveis aleatórias independentes. O Exercício 9.2-2 pede que você justifique essa afirmação.

Vamos considerar a expressão $\max(k-1, n-k)$. Temos

$$\max(k-1, n-k) = \begin{cases} k-1 \text{ se } k > \lceil n/2 \rceil, \\ n-k \text{ se } k \le \lceil n/2 \rceil. \end{cases}$$

Se n é par, cada termo de $T(n/2)$ até $T(n-1)$ aparece exatamente duas vezes no somatório, e, se n é ímpar, todos esses termos aparecem duas vezes e o termo $T(n/2)$ aparece uma vez. Assim, temos

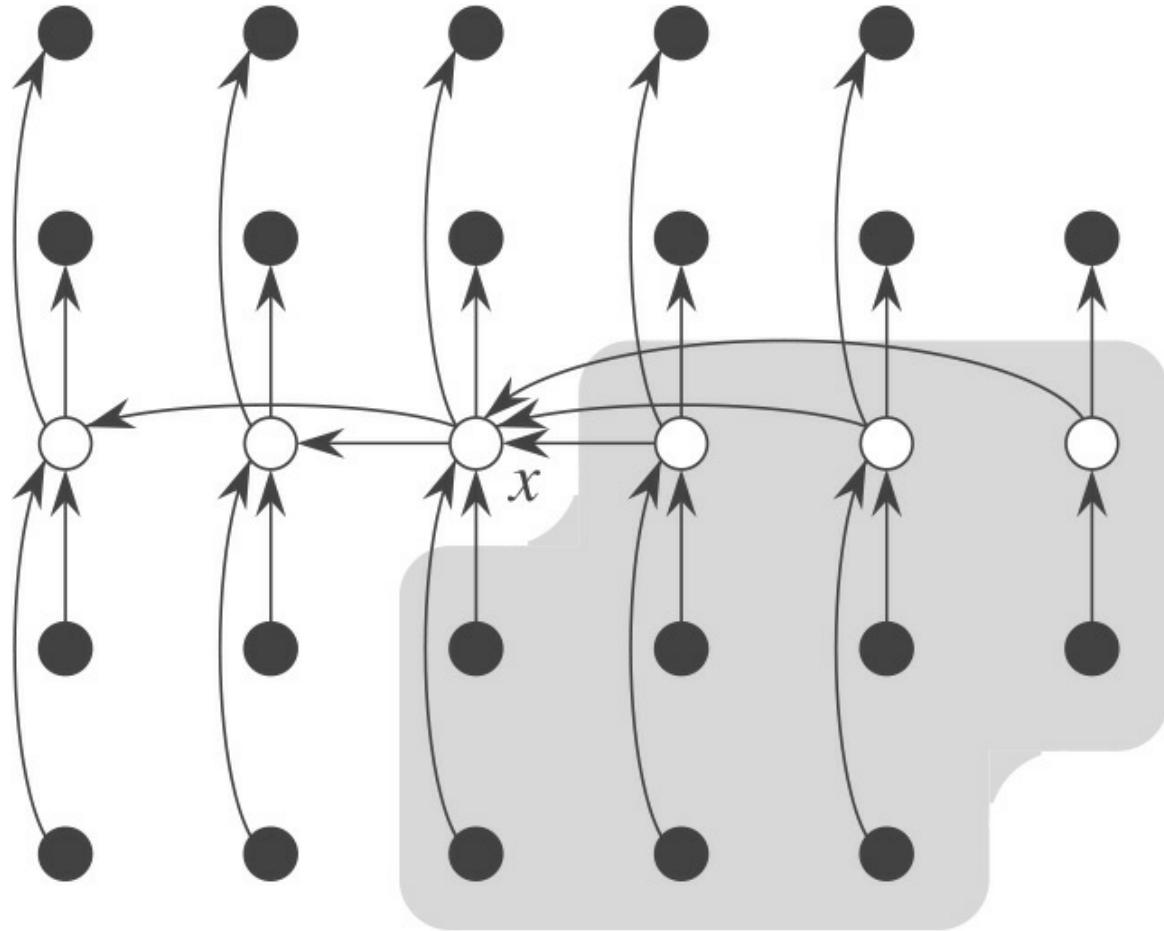
$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n)$$

Mostramos que $E[T(n)] = O(n)$ por substituição. Suponha que $E[T(n)] \leq cn$ para alguma constante c que satisfaça as condições iniciais da recorrência. Supomos $T(n) = O(1)$ para n menor que alguma constante; escolheremos essa constante mais adiante. Também escolhemos uma constante a tal que a função descrita pelo termo $O(n)$ citado (que descreve o componente não recursivo do tempo de execução do algoritmo) seja limitada acima por an para todo $n > 0$. Usando essa hipótese de indução, temos

$$\begin{aligned}
E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor -1} k \right) + an \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\
&\leq \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\
&= \frac{2c}{n} \left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\
&= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\
&= c \left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\
&\leq \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right).
\end{aligned}$$

Para concluir a prova, precisamos mostrar que, para n suficientemente grande, esta última expressão é no máximo cn ou, o que é equivalente, $cn/4 - c/2 - an \geq 0$. Se somarmos $c/2$ a ambos os lados e fatorarmos n, obteremos $n(c/4 - a) \geq c/2$. Desde que escolhamos a constante c de modo que $c/4 - a > 0$, isto é, $c > 4a$, poderemos dividir ambos os lados por $c/4 - a$, obtendo

$$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}.$$

Assim, se considerarmos que $T(n) = O(1)$ para $n < 2c/(c - 4a)$, então $E[T(n)] = O(n)$. Concluímos que, podemos determinar qualquer estatística de ordem, e em particular a mediana, em tempo linear esperado, considerando que os elementos são distintos.

Exercícios

9.2-1 Mostre que RANDOMIZED-SELECT nunca faz uma chamada recursiva a um arranjo de comprimento 0.

9.2-2 Demonstre que a variável aleatória indicadora X_k e o valor $T(\max(k - 1, n - k))$ são independentes.

9.2-3 Escreva uma versão iterativa de RANDOMIZED-SELECT.

9.2-4 Suponha que usamos RANDOMIZED-SELECT para selecionar o elemento mínimo do arranjo $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Descreva uma sequência de partições que resulte em um desempenho do pior caso de RANDOMIZED-SELECT.

9.3 SELEÇÃO EM TEMPO LINEAR DO PIOR CASO

Examinaremos agora um algoritmo de seleção cujo tempo de execução é $O(n)$ no pior caso. Como RANDOMIZED-SELECT, o algoritmo SELECT determina o elemento desejado particionando recursivamente o arranjo de entrada. Todavia, aqui *garantimos* uma boa divisão particionando o arranjo. SELECT utiliza o algoritmo de particionamento determinístico PARTITION do quicksort (veja a Seção 7.1), porém modificado para tomar o elemento em torno do qual é executada a partição como um parâmetro de entrada.

Figura 9.1 Análise do algoritmo SELECT. Os n elementos são representados por pequenos círculos, e cada grupo de cinco elementos ocupa uma coluna. As medianas dos grupos são os círculos brancos e a mediana de medianas x é identificada. (Para determinar a mediana de um número par de elementos, usamos a mediana inferior.) As setas são orientadas dos elementos maiores para os menores e, com isso, podemos ver que três em cada grupo completo de cinco elementos à direita de x são maiores que x, e três em cada grupo de cinco elementos à esquerda de x são menores que x. Os elementos maiores que x são mostrados sobre um fundo sombreado.

O algoritmo SELECT determina o i-ésimo menor elemento de um arranjo de entrada de $n > 1$ elementos distintos executando as etapas a seguir. (Se $n = 1$, então SELECT simplesmente retorna seu único valor de entrada como o i-ésimo menor.)

1. Dividir os n elementos do arranjo de entrada em $n/5$ grupos de cinco elementos cada e no máximo um grupo formado pelos n mod 5 elementos restantes.
2. Determinar a mediana de cada um dos $n/5$ grupos, primeiro ordenando por inserção os elementos de cada grupo (dos quais existem cinco, no máximo) e depois escolhendo a mediana na lista ordenada de elementos de grupos.
3. Usar SELECT recursivamente para definir a mediana x das $n/5$ medianas determinadas na etapa 2. (Se houver um número par de medianas, pela nossa convenção, x é a mediana inferior.)
4. Particionar o arranjo de entrada em torno da mediana de medianas x usando a versão modificada de PARTITION. Seja k um mais que o número de elementos no lado baixo da partição, de modo que x é o k-ésimo menor elemento e há $n - k$ elementos no lado alto da partição.
5. Se $i = k$, então retornar x. Caso contrário, usar SELECT recursivamente para determinar o i-ésimo menor elemento no lado baixo se $i \leq k$ ou o $(i - k)$-ésimo menor elemento no lado alto se $i > k$.

Para analisar o tempo de execução de SELECT, primeiro determinamos um limite inferior para o número de elementos que são maiores que o elemento de particionamento x. A Figura 9.1 nos ajuda a visualizar essa contabilidade. No mínimo, metade das medianas determinadas na etapa 2 é maior ou igual à mediana de medianas x.[1] Assim, no mínimo metade dos $n/5$ grupos contribuem com no mínimo três elementos maiores que x, exceto o único grupo que tem menos de cinco elementos se 5 não for um divisor exato de n, e o único grupo que contém o próprio x. Descontando esses dois grupos, decorre que o número de elementos maiores que x é, no mínimo,

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{2}\right\rceil\right\rceil - 2\right) \geq \frac{3n}{10} - 6 .$$

De modo semelhante, no mínimo $3n/10 - 6$ elementos são menores que x. Assim, no pior caso, a etapa 5 chama SELECT recursivamente para, no máximo, $7n/10 + 6$ elementos.

Agora podemos desenvolver uma recorrência para o tempo de execução do pior caso $T(n)$ do algoritmo SELECT. As etapas 1, 2 e 4 demoram o tempo $O(n)$. (A etapa 2 consiste em $O(n)$ chamadas de ordenação por inserção para conjuntos de tamanho $O(1)$.) A etapa 3 demora o tempo $T(n/5)$, e a etapa 5 demora no máximo o tempo $T(7n/10 + 6)$ considerando que T é monotonicamente crescente. Adotamos a seguinte premissa, que a princípio parece sem motivo: qualquer entrada com menos de 140 elementos requer o tempo $O(1)$; a origem da constante mágica 140 ficará clara em breve. Portanto, podemos obter a recorrência

$$T(n) \leq \begin{cases} O(1) & se\ n < 140, \\ T(\lceil n/5 \rceil) T(7n/10 + 6) + O(n) & se\ n \geq 140. \end{cases}$$

Mostramos que o tempo de execução é linear por substituição. Mais especificamente, mostraremos que $T(n) \leq cn$ para alguma constante c adequadamente grande e para todo $n > 0$. Começamos supondo que $T(n) \leq cn$ para alguma constante c adequadamente grande e para todo $n \leq 140$; essa premissa se mantém válida se c for suficientemente grande. Também escolhemos uma constante a tal que a função descrita pelo termo $O(n)$ citado (que descreve o componente não recursivo do tempo de execução do algoritmo) é limitado acima por an para todo $n > 0$. Substituindo essa hipótese indutiva no lado direito da recorrência, obtemos

$$\begin{aligned} T(n) &\le c\lceil n/5\rceil + c(7n/10+6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

que é no máximo cn se

$$-cn/10 + 7c + an \le 0\ . \tag{9.2}$$

A desigualdade (9.2) é equivalente à desigualdade $c \ge 10a(n/(n-70))$ quando $n > 70$. Como admitimos que $n \ge 140$, temos $n/(n-70) \le 2$ e, assim, escolher $c \ge 20a$ satisfará a desigualdade (9.2). (Observe que não há nada de especial na constante 140; poderíamos substituí-la por qualquer inteiro estritamente maior que 70 e depois escolher c de acordo.) Então, o tempo de execução do pior caso de SELECT é linear.

Como em uma ordenação por comparação (veja a Seção 8.1), SELECT e RANDOMIZED-SELECT determinam informações sobre a ordem relativa de elementos somente por comparação de elementos. Vimos no Capítulo 8 que a ordenação exige o tempo $\Omega(n \lg n)$ no modelo de comparação, mesmo na média (veja o Problema 8-1). Os algoritmos de ordenação de tempo linear do Capítulo 8 adotam premissas sobre a entrada. Ao contrário, os algoritmos de seleção de tempo linear deste capítulo não exigem nenhuma premissa sobre a entrada. Eles não estão sujeitos ao limite inferior $(n \lg n)$ porque conseguem resolver o problema da seleção sem ordenar. Assim, resolver o problema da seleção por ordenação e indexação, como apresentamos no início deste capítulo, é assintoticamente ineficiente.

Exercícios

9.3-1 No algoritmo SELECT, os elementos de entrada são divididos em grupos de cinco. O algoritmo funcionará em tempo linear se eles forem divididos em grupos de sete? Demonstre que SELECT não será executado em tempo linear se forem usados grupos de três elementos.

9.3-2 Analise SELECT para mostrar que, se $n \ge 140$, então no mínimo $n/4$ elementos são maiores que a mediana de medianas x e no mínimo $n/4$ elementos são menores que x.

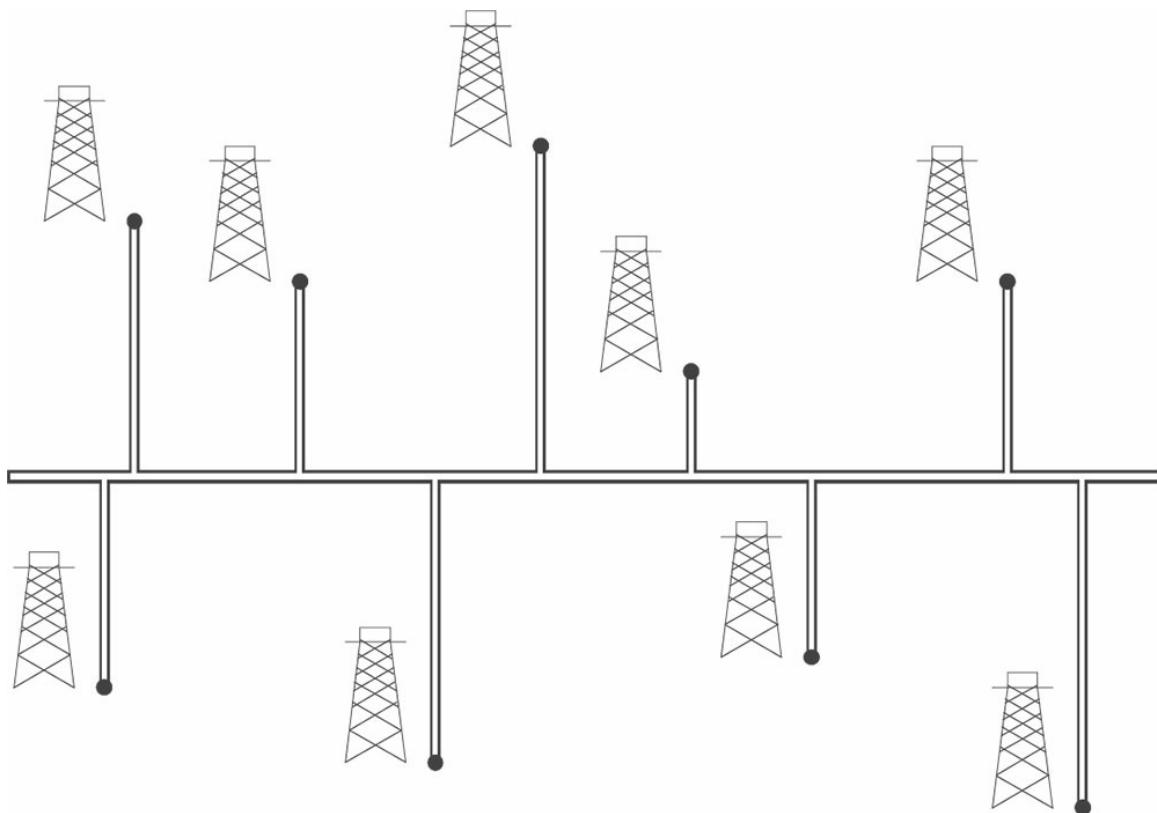
9.3-3 Mostre como o quicksort pode ser modificado para ser executado no tempo $O(n \lg n)$ no pior caso, considerando que todos os elementos são distintos.

9.3-4 ★ Suponha que um algoritmo utilize somente comparações para determinar o i-ésimo menor elemento em um conjunto de n elementos. Mostre que ele também pode determinar os $i-1$ menores elementos e os $n-i$ maiores elementos sem executar quaisquer comparações adicionais.

9.3-5 Dada uma sub-rotina "caixa-preta" para a mediana de tempo linear no pior caso, apresente um algoritmo simples de tempo linear que resolva o problema de seleção para uma estatística de ordem arbitrária.

9.3-6 Os k-ésimos ***quantis*** de um conjunto de n elementos são as $k-1$ estatísticas de ordem que dividem o conjunto ordenado em k conjuntos de igual tamanho (com aproximação de 1). Apresente um algoritmo de tempo $O(n \lg k)$ para dar uma lista dos k-ésimos quantis de um conjunto.

9.3-7 Descreva um algoritmo de tempo $O(n)$ que, dados um conjunto S de n números distintos e um inteiro positivo $k \le n$, determine os k números em S que estão mais próximos da mediana de S.

9.3-8 Sejam $X[1 \mathinner{..} n]$ e $Y[1 \mathinner{..} n]$ dois arranjos, cada um contendo n números já em sequência ordenada. Apresente um algoritmo de tempo $O(\lg n)$ para determinar a mediana de todos os $2n$ elementos nos arranjos X e Y.

9.3-9 O professor Olay é consultor de uma empresa petrolífera que está planejando um grande oleoduto de leste para oeste que atravessa um campo petrolífero com n poços. A empresa quer conectar um duto auxiliar entre cada um desses poços e o oleoduto principal ao longo de um caminho mais curto (para o norte ou para o sul), como mostra a Figura 9.2. Dadas as coordenadas x e y dos poços, como o professor deve escolher a localização ótima do oleoduto principal que minimizará o comprimento total dos dutos auxiliares? Mostre como determinar a localização ótima em tempo linear.

Problemas

9-1 ***Os* i *maiores números em sequência ordenada***

Dado um conjunto de n números, queremos determinar os i maiores em sequência ordenada usando um algoritmo baseado em comparação. Descubra o algoritmo que implementa cada um dos métodos a seguir com o melhor tempo de execução assintótico do pior caso e analise os tempos de execução dos algoritmos em termos de n e i.

Figura 9.2 O professor Olay precisa determinar a posição do oleoduto de leste para oeste que minimiza o comprimento total dos dutos auxiliares norte-sul.

a. Classifique os números e produza uma lista com os i maiores.

b. Construa uma fila de prioridade máxima com os números e chame EXTRACT-MAX i vezes.

c. Use um algoritmo de estatística de ordem para determinar o *i*-ésimo maior número, particionar em torno desse número e ordenar os *i* maiores números.

9-2 Mediana ponderada

Para n elementos distintos $x_1, x_2, ..., x_n$, com pesos positivos $w_1, w_2, ..., w_n$ tais que $\sum_{i=1}^{n} w_i = 1$, a ***mediana ponderada*** (***inferior***) é o elemento x^k que satisfaz

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

e

$$\sum_{x_i > x_k} w_i \le \frac{1}{2}$$

Por exemplo, se os elementos são 0,1; 0,35; 0,05; 0,1; 0,15; 0,05; 0,2 e cada elemento é igual ao seu peso (isto é, $w_i = x_i$ para $i = 1, 2, ..., 7$), então a mediana é 0,1, mas a mediana ponderada é 0,2.

a. Mostre que a mediana de $x_1, x_2, ..., x_n$ é a mediana ponderada dos x_i com pesos $w_i = 1/n$ para $i = 1, 2, ..., n$.

b. Mostre como calcular a mediana ponderada de n elementos no tempo $O(n \lg n)$ do pior caso usando ordenação.

c. Mostre como calcular a mediana ponderada no tempo Q(n) do pior caso usando um algoritmo de mediana de tempo linear como SELECT da Seção 9.3.

O ***problema da localização da agência postal*** é definido da seguinte maneira: temos n pontos $p_1, p_2, ..., p_n$ com pesos associados $w_1, w_2, ..., w_n$. Desejamos determinar um ponto p (não necessariamente um dos pontos de entrada) que minimize o somatório $\sum_{i=1}^{n} w_i d(p, p_i)$, onde $d(a, b)$ é a distância entre os pontos a e b.

d. Mostre que a mediana ponderada é uma solução melhor para o problema da localização de agência postal unidimensional, no qual os pontos são simplesmente números reais e a distância entre os pontos a e b é $d(a, b) = |a - b|$.

e. Determine a melhor solução para o problema de localização da agência postal bidimensional, no qual os pontos são pares de coordenadas (x, y) e a distância entre os pontos $a = (x_1, y_1)$ e $b = (x_2, y_2)$ é a ***distância Manhattan*** dada por $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9-3 Estatísticas de ordem pequena

Mostramos que o número $T(n)$ de comparações do pior caso usadas por SELECT para selecionar a *i*-ésima estatística de ordem de n números satisfaz a $T(n)$ = Q(n), mas a constante oculta pela notação Q é bastante grande. Quando i é pequeno em relação a n, podemos implementar um procedimento diferente que utiliza SELECT como uma sub-rotina, mas executa um número menor de comparações no pior caso.

a. Descreva um algoritmo que utilize $U_i(n)$ comparações para determinar o *i*-ésimo menor de n elementos, onde

$$U_i(n) = \begin{cases} T(n) & \text{se } i \geq n/2, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{caso contrário.} \end{cases}$$

(*Sugestão*: Comece com $n/2$ comparações disjuntas aos pares e efetue a recursão sobre o conjunto que contém o menor elemento de cada par.)

b. Mostre que, se $i < n/2$, então $U_i(n) = n + O(T(2i) \lg(n/i))$.

c. Mostre que, se i é uma constante menor que $n/2$, então $U_i(n) = n + O(\lg n)$.

d. Mostre que, se $i = n / k$ para $k \geq 2$, então $U_i(n) = n + O(T(2n/k) \lg k)$.

1-4 ***Análise alternativa da seleção aleatorizada***

Neste problema, usamos variáveis aleatórias indicadoras para analisar o procedimento RANDOMIZED-SELECT de um modo semelhante ao de nossa análise de RANDOMIZED-QUICKSORT na Seção 7.4.2.

Como na análise do quicksort, admitimos que todos os elementos são distintos e renomeanos os elementos do conjunto de entrada como A as $z_1, z_2,..., z_n$ onde z_i é o i-ésimo menor elemento. Assim, a chamada RANDOMIZED-SELECT $(A, 1, n, k)$ retorna z_k.

Para $1 \leq i < j \leq n$, seja

X_{ijk} = I {z_i é comparado com z_j em algum momento durante a execução do algoritmo para determinar z_k}.

a. Dê uma expressão exata para E X_{ijk}. (*Sugestão:* Sua expressão pode ter valores diferentes, dependendo dos valores de i, j e k.)

b. Represente por X_k o número total de comparações entre elementos do arranjo A ao determinar z_k. Mostre que

$$E[X_k] \leq 2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1} + \sum_{j=k+1}^{n}\frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}\right)$$

c. Mostre que $E[X_k] \leq 4n$.

d. Conclua que, considerando que todos os elementos do arranjo A são distintos, RANDOMIZED-SELECT é executado no tempo esperado $O(n)$.

NOTAS DO CAPÍTULO

O algoritmo de tempo linear do pior caso para determinar a mediana foi criado por Blum, Floyd, Pratt, Rivest e Tarjan [50]. A versão aleatorizada rápida se deve a Hoare [169]. Floyd e Rivest [108] desenvolveram uma versão aleatorizada melhorada que particiona em torno de um elemento selecionado recursivamente de uma pequena amostra dos elementos.

Ainda não se sabe exatamente quantas comparações são necessárias para determinar a mediana. Bent e John [41] deram um limite inferior de $2n$ comparações para determinar a mediana, e Schönhage, Paterson e Pippenger [302] deram um limite superior de $3n$. Dor e Zwick melhoraram esses limites. O limite superior dado por eles [93] é ligeiramente menor que $2{,}95n$ e o limite inferior [94] é $(2 +)\ n$, para uma pequena constante positiva , o que é uma

ligeira melhoria do trabalho relacionado de Dor et al. [92]. Paterson [272] descreve alguns desses resultados juntamente com outro trabalho relacionado.

---

1 Como admitimos que os números são distintos, todas as medianas, exceto x, são maiores ou menores que x.

Parte

ESTRUTURAS DE DADOS

INTRODUÇÃO

Conjuntos são tão fundamentais para a Ciência da Computação quanto para a Matemática. Enquanto os conjuntos matemáticos são invariáveis, os conjuntos manipulados por algoritmos podem crescer, encolher ou sofrer outras mudanças ao longo do tempo. Chamamos tais conjuntos de conjuntos ***dinâmicos***. Os próximos cinco capítulos apresentam algumas técnicas básicas para representar conjuntos dinâmicos finitos e para manipular esses conjuntos em um computador.

Algoritmos podem exigir a execução de vários tipos diferentes de operações em conjuntos. Por exemplo, muitos algoritmos precisam apenas da capacidade de inserir e eliminar elementos em um conjunto e testar a pertinência de elementos a um conjunto. Damos o nome de ***dicionário*** ao conjunto dinâmico que suporta essas operações. Outros algoritmos exigem operações mais complicadas. Por exemplo, filas de prioridade mínima, que foram introduzidas no Capítulo 6 no contexto da estrutura de dados heap, suportam as operações de inserção de um elemento no conjunto e de extração do menor elemento de um conjunto. A melhor maneira de implementar um conjunto dinâmico depende das operações que devem ser suportadas.

Elementos de um conjunto dinâmico

Em uma implementação típica de um conjunto dinâmico, cada elemento é representado por um objeto cujos atributos podem ser examinados e manipulados se tivermos um ponteiro para o objeto (a Seção 10.3 discute a implementação de objetos e ponteiros em ambientes de programação que não os contêm como tipos de dados básicos). Alguns tipos de conjuntos dinâmicos consideram que um dos atributos do objeto é uma ***chave*** de identificação. Se as chaves são todas diferentes, podemos imaginar o conjunto dinâmico como um conjunto de valores de chaves. O objeto pode conter ***dados satélites***, que são transportados em atributos de outro objeto mas que, fora isso, não são utilizados pela implementação do conjunto. Também pode ter atributos que são manipulados pelas operações de conjuntos; esses atributos podem conter dados ou ponteiros para outros objetos no conjunto.

Alguns conjuntos dinâmicos pressupõem que as chaves são extraídas de um conjunto totalmente ordenado como o dos números reais ou o de todas as palavras sob a ordenação alfabética usual. Uma ordenação total nos permite definir o elemento mínimo do conjunto, por exemplo, ou falar do próximo elemento maior que um dado elemento em um conjunto.

Operações em conjuntos dinâmicos

As operações em um conjunto dinâmico podem ser agrupadas em duas categorias: ***consultas***, que simplesmente retornam informações sobre o conjunto, e ***operações modificadoras***, que alteram o conjunto. Apresentamos a seguir,

uma lista de operações típicas. Qualquer aplicação específica, normalmente exigirá a implementação de apenas algumas dessas operações.

SEARCH(S, k)

Uma consulta que, dado um conjunto S e um valor de chave k, retorna um ponteiro x para um elemento em S tal que $x.chave = k$ ou NIL se nenhum elemento desse tipo pertencer a S.

INSERT(S, x)

Uma operação modificadora que aumenta o conjunto S com o elemento apontado por x. Normalmente, consideramos que quaisquer atributos no elemento x necessários para a implementação do conjunto já foram inicializados.

DELETE(S, x)

Uma operação modificadora que, dado um ponteiro x para um elemento no conjunto S, remove x de S. (Observe que essa operação utiliza um ponteiro para um elemento x, não um valor de chave.)

MININUM(S)

Uma consulta em um conjunto totalmente ordenado S que retorna um ponteiro para o elemento de S que tenha a menor chave.

MAXIMUM(S)

Uma consulta em um conjunto totalmente ordenado S que retorna um ponteiro para o elemento de S que tenha a maior chave.

SUCCESSOR(S, x)

Uma consulta que, dado um elemento x cuja chave é de um conjunto totalmente ordenado S, retorna um ponteiro para o elemento maior seguinte em S ou NIL se x é o elemento máximo.

PREDECESSOR(S, x)

Uma consulta que, dado um elemento x, cuja chave é de um conjunto totalmente ordenado S, >retorna um ponteiro para o elemento menor seguinte em S ou NIL se x é o elemento mínimo.

Em algumas situações, podemos estender as consultas SUCCESSOR e PREDECESSOR de modo que se apliquem a conjuntos com chaves não distintas. Para um conjunto com n chaves, normalmente presume-se que uma chamada a MINIMUM seguida por $n - 1$ chamadas a SUCCESSOR enumera os elementos no conjunto em sequência ordenada.

Em geral, medimos o tempo empregado para executar uma operação de conjunto em termos do tamanho do conjunto. Por exemplo, o Capítulo 13 descreve uma estrutura de dados que pode suportar qualquer das operações da lista apresentada em um conjunto de tamanho n no tempo $O(\lg n)$.

Visão geral da Parte III

Os Capítulos 10 a 14 descrevem várias estruturas de dados que podemos usar para implementar conjuntos dinâmicos; mais adiante, usaremos muitas dessas estruturas para construir algoritmos eficientes para uma variedade de problemas. Já vimos uma outra estrutura de dados importante — o heap — no Capítulo 6.

O Capítulo 10 apresenta os fundamentos essenciais do trabalho com estruturas de dados simples como pilhas, filas, listas ligadas e árvores enraizadas. Mostra também como implementar objetos e ponteiros em ambientes de programação que não os suportam como primitivas. Se você frequentou um curso introdutório de programação, já deve estar familiarizado com grande parte desse material.

O Capítulo 11 introduz as tabelas hash (ou tabelas de espalhamento) que suportam as operações de dicionário INSERT, DELETE e SEARCH. No pior caso, o hashing (ou espalhamento) requer o tempo $Q(n)$ para executar uma operação SEARCH, mas o tempo esperado para operações de tabelas de espalhamento é $O(1)$. A análise do hashing se baseia na probabilidade, mas a maior parte do capítulo não requer nenhuma experiência no assunto.

As árvores de busca binária, que são focalizadas no Capítulo 12, suportam todas as operações de conjuntos dinâmicos que figuram na lista apresentada. No pior caso, cada operação demora o tempo $Q(n)$ em uma árvore com n elementos mas, em uma árvore de busca binária construída aleatoriamente, o tempo esperado para cada operação é $O(\lg n)$. As árvores de busca binária servem como base para muitas outras estruturas de dados.

O Capítulo 13 apresenta as árvores vermelho-preto, que são uma variante de árvores de busca binária. Diferentemente das árvores de busca binária comuns, o bom funcionamento das árvores vermelho-preto é garantido: as operações demoram o tempo $O(\lg n)$ no pior caso. Uma árvore vermelho-preto é uma árvore de busca balanceada; o Capítulo 18 na Parte V apresenta um outro tipo de árvore de busca balanceada, denominada árvore B. Embora a mecânica das árvores vermelho-preto seja um pouco complicada, você pode perceber grande parte de suas propriedades pelo capítulo, sem estudar detalhadamente a mecânica. Não obstante, você verá que examinar o código é bastante instrutivo.

No Capítulo 14, mostramos como aumentar as árvores vermelho-preto para suportar outras operações além das básicas que aparecem na lista apresentada. Primeiro, aumentamos as árvores de modo que possamos manter dinamicamente estatísticas de ordem para um conjunto de chaves. Em seguida, nós as aumentamos de um modo diferente para manter intervalos de números reais.

10 ESTRUTURAS DE DADOS ELEMENTARES

Neste capítulo, examinaremos a representação de conjuntos dinâmicos por estruturas de dados simples que usam ponteiros. Embora possamos construir muitas estruturas de dados complexas que utilizam ponteiros, apresentaremos apenas as rudimentares: pilhas, filas, listas ligadas e árvores enraizadas. Também mostraremos meios de sintetizar objetos e ponteiros partindo de arranjos.

10.1 PILHAS E FILAS

Pilhas e filas são conjuntos dinâmicos nos quais o elemento removido do conjunto pela operação DELETE é especificado previamente. Em uma ***pilha***, o elemento eliminado do conjunto é o mais recentemente inserido: a pilha implementa uma política de ***último a entrar***, ***primeiro a sair*** ou ***LIFO*** (last-in, first-out). De modo semelhante, em uma ***fila*** o elemento eliminado é sempre o que estava no conjunto há mais tempo: a fila implementa uma política de ***primeiro a entrar***, ***primeiro a sair*** ou ***FIFO*** (first-in, first-out). Há vários modos eficientes de implementar pilhas e filas em um computador. Nesta seção, mostraremos como usar um arranjo simples para implementar cada uma delas.

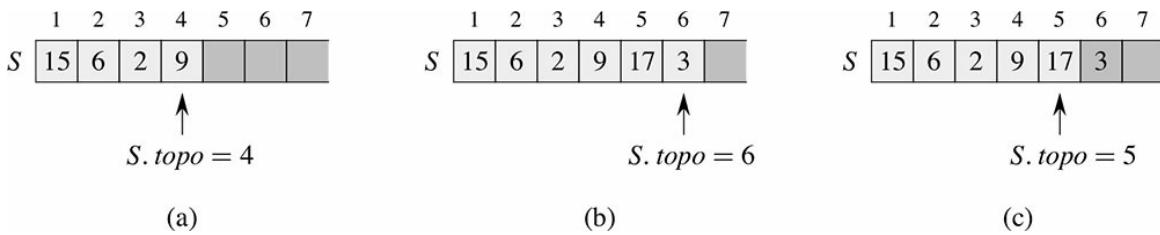
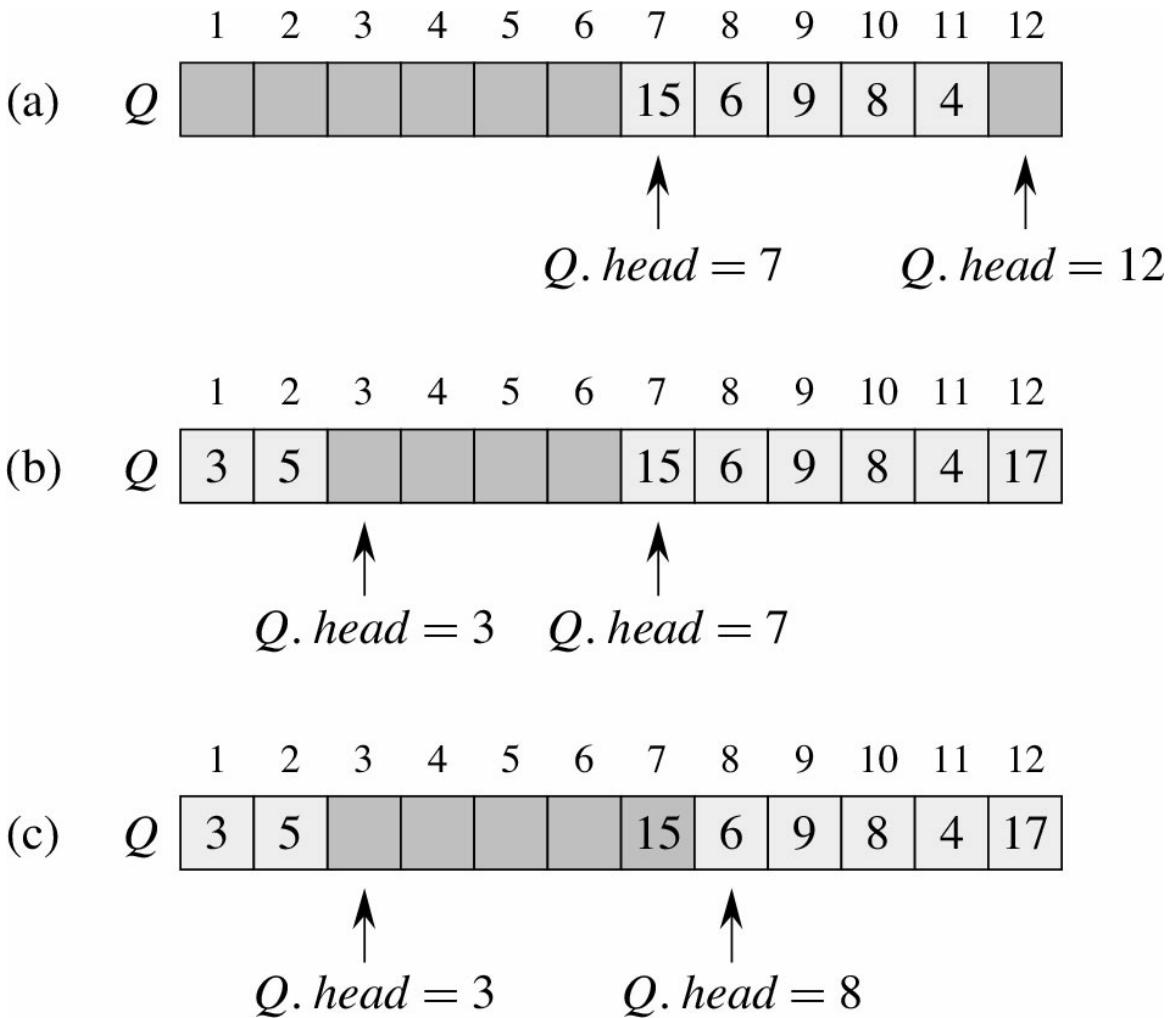
Pilhas

A operação INSERT em uma pilha é frequentemente denominada PUSH, e a operação DELETE, que não toma um argumento de elemento, é frequentemente denominada POP. Esses nomes são alusões a pilhas físicas, como as pilhas de pratos acionadas por molas usadas em restaurantes. A ordem em que os pratos são retirados da pilha é o inverso da ordem em que foram colocados na pilha, já que apenas o prato do topo está acessível.

Como mostra a Figura 10.1, podemos implementar uma pilha de no máximo n elementos com um arranjo $S[1..\,n]$. O arranjo tem um atributo $S.topo$ que indexa o elemento mais recentemente inserido. A pilha consiste nos elementos $S[1\,..\,S.topo]$, onde $S[1]$ é o elemento na parte inferior da pilha e $S[S.topo]$ é o elemento na parte superior.

Quando $S.topo = 0$, a pilha não contém nenhum elemento e está ***vazia***. Podemos testar se a pilha está vazia pela operação de consulta STACK-EMPTY. Se tentarmos extrair algo de uma pilha vazia, dizemos que a pilha tem ***estouro negativo***, que é normalmente um erro. Se $S.topo$ exceder n, a pilha tem um ***estouro***. (Em nossa implementação de pseudocódigo, não nos preocuparemos com estouro de pilha.)

Podemos implementar cada uma das operações em pilhas com apenas algumas linhas de código.

```
STACK-EMPTY(S)
1    if S.topo == 0
2        return TRUE
3    else return FALSE

PUSH(S, x)
1    S.topo = S.topo + 1
2    S[S.topo] = x

POP(S)
1    if STACK-EMPTY(S)
2        error "underflow"
3    else S.topo = S.topo − 1
4        return S[S.topo + 1]
```

A Figura 10.1 mostra os efeitos das operações modificadoras PUSH (EMPILHAR) e POP (DESEMPILHAR). Cada uma das três operações em pilha demora o tempo $O(1)$.

Filas

Designamos a operação INSERT em uma fila por ENQUEUE (ENFILEIRAR) e a operação DELETE por DEQUEUE (DESINFILEIRAR); assim como a operação em pilhas POP, DEQUEUE não adota nenhum argumento de elemento. A propriedade FIFO de uma fila faz com que ela funcione como uma fileira de pessoas em uma caixa registradora. A fila tem um ***início*** (ou cabeça) e um ***fim*** (ou cauda). Quando um elemento é inserido na fila, ocupa seu lugar no fim da fila, exatamente como um cliente que acabou de chegar ocupa um lugar no final da fileira. O elemento retirado da fila é sempre aquele que está no início da fila, como o cliente que está no início da fileira e esperou por mais tempo.

A Figura 10.2 mostra um modo de implementar uma fila de no máximo $n - 1$ elementos usando um arranjo $Q[1 .. n]$. A fila tem um atributo $Q.início$ que indexa ou aponta para seu início. O atributo $Q.fim$ indexa a próxima posição na qual um elemento recém-chegado será inserido na fila. Os elementos na fila estão nas posições $Q.início$, $Q.início + 1$, ..., $Q.fim - 1$, onde "retornamos", no sentido de que a posição 1 segue imediatamente a posição n em uma ordem circular. Quando $Q.início = Q.fim$, a fila está vazia. Inicialmente, temos $Q.início = Q.fim = 1$. Se tentarmos desenfileirar um elemento de uma fila vazia, a fila sofre perda de dígitos. Quando $Q.início = Q.fim + 1$ ou simultaneamente $Q.início = 1$ e $Q.fim = Q.comprimento$, a fila está cheia e, se tentarmos enfileirar um elemento, a fila sofre estouro.

Figura 10.1 Uma implementação de arranjo de uma pilha S. Os elementos da pilha aparecem somente nas posições sombreadas em tom mais claro. (**a**) A pilha S tem quatro elementos. O elemento do topo é 9. (**b**) A pilha S após as chamadas PUSH(S, 17) e PUSH(S, 3). (**c**) A pilha S após a chamada POP(S) retornou o elemento 3, que é o elemento mais recentemente inserido na pilha. Embora ainda apareça no arranjo, o elemento 3 não está mais na pilha; o topo é o elemento 17.

Figura 10.2 Uma fila implementada com a utilização de um arranjo Q 1 .. 12. Os elementos da fila aparecem somente nas posições sombreadas em tom mais claro. (**a**) A fila tem cinco elementos, nas posições Q 7 .. 11. **(b)**A configuração da fila após as chamadas EnquEuE(Q, 17), EnquEuE(Q, 3) e EnquEuE(Q, 5). (**c**) A configuração da fila após a chamada DEquEuE(Q) retornar o valor de chave 15 que se encontrava anteriormente no início da fila. O novo início tem chave 6.

Em nossos procedimentos Enqueue e Dequeue, omitimos a verificação de erros de estouro negativo e estouro. (O Exercício 10.1-4 pede que você forneça o código que verifica essas duas condições de erro.) O pseudocódigo considera que n = $Q.comprimento$.

Enqueue(Q, x)

1 $Q[Q.fim] = x$
2 **if** $Q.fim = Q.comprimento$
3 $\quad Q.fim = 1$
4 **else** $Q.fim = Q.fim + 1$

Dequeue(Q)

1 $x = Q[Q.início]$
2 **if** $Q.início == Q.comprimento$
3 $\quad Q.início = 1$
4 **else** $Q.início = Q.início + 1$
5 **return** x

A Figura 10.2 mostra os efeitos das operações ENQUEUE e DEQUEUE. Cada operação demora o tempo $O(1)$.

Exercícios

10.1-1 Usando a Figura 10.1 como modelo, ilustre o resultado de cada operação na sequência PUSH(S, 4), PUSH(S, 1), PUSH(S, 3), POP(S), PUSH(S, 8) e POP(S) sobre uma pilha S inicialmente vazia armazenada no arranjo $S[1 .. 6]$.

10.1-2 Explique como implementar duas pilhas em um único arranjo $A[1 .. n]$ de tal modo que nenhuma delas sofra um estouro a menos que o número total de elementos em ambas as pilhas juntas seja n. As operações PUSH e POP devem ser executadas no tempo $O(1)$.

10.1-3 Usando a Figura 10.2 como modelo, ilustre o resultado de cada operação na sequência ENQUEUE(Q, 4), ENQUEUE(Q, 1), ENQUEUE(Q, 3), DEQUEUE(Q), ENQUEUE(Q, 8) e DEQUEUE(Q) em uma fila Q inicialmente vazia armazenada no arranjo $Q[1 .. 6]$.

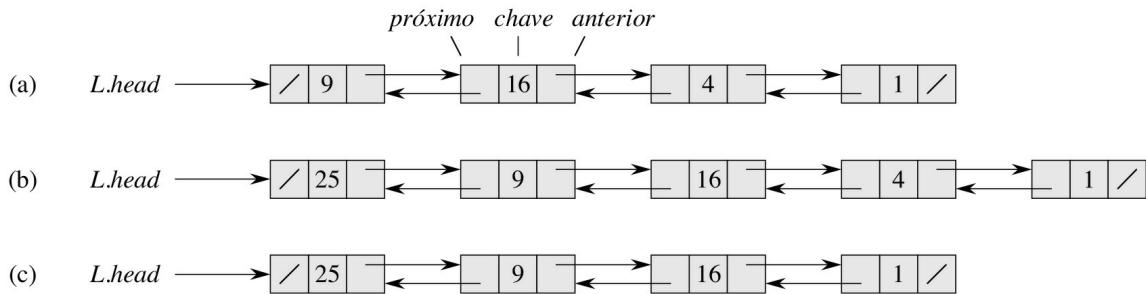
10.1-4 Reescreva ENQUEUE e DEQUEUE para detectar o estouro negativo e o estouro de uma fila.

10.1-5 Enquanto uma pilha permite inserção e eliminação de elementos em apenas uma extremidade e uma fila permite inserção em uma extremidade e eliminação na outra extremidade, uma ***deque*** (double-ended queue, ou fila de extremidade dupla) permite inserção e eliminação em ambas as extremidades. Escreva quatro procedimentos de tempo $O(1)$ para inserir elementos e eliminar elementos de ambas as extremidades de uma deque construída a partir de um arranjo.

10.1-6 Mostre como implementar uma fila usando duas pilhas. Analise o tempo de execução das operações em filas.

10.1-7 Mostre como implementar uma pilha usando duas filas. Analise o tempo de execução das operações em pilhas.

10.2 LISTAS LIGADAS

Uma ***lista ligada*** é uma estrutura de dados na qual os objetos estão organizados em ordem linear. Entretanto, diferentemente de um arranjo, no qual a ordem linear é determinada pelos índices do arranjo, a ordem em uma lista ligada é determinada por um ponteiro em cada objeto. Listas ligadas nos dão uma representação simples e flexível para conjuntos dinâmicos, suportando (embora não necessariamente com eficiência) todas as operações que aparecem na lista à página 166.

Como mostra a Figura 10.3, cada elemento de uma ***lista duplamente ligada*** L é um objeto com um atributo *chave* e dois outros atributos *ponteiros*: *próximo* e *anterior*. O objeto também pode conter outros dados satélites. Dado um elemento x na lista, $x.próximo$ aponta para seu sucessor na lista ligada e $x.anterior$ aponta para seu predecessor. Se $x.anterior$ = NIL, o elemento x não tem nenhum predecessor e, portanto, é o primeiro elemento, ou ***início***, da lista. Se $x.próximo$ = NIL, o elemento x não tem nenhum sucessor e, assim, é o último elemento, ou ***fim***, da lista. Um atributo $L.início$ aponta para o primeiro elemento da lista. Se $L.início$ = NIL, a lista está vazia.

Uma lista pode ter uma entre várias formas. Ela pode ser simplesmente ligada ou duplamente ligada, pode ser ordenada ou não e pode ser circular ou não. Se uma lista é ***simplesmente ligada***, omitimos o ponteiro *anterior* em cada elemento. Se uma lista é ***ordenada***, a ordem linear da lista corresponde à ordem linear de chaves armazenadas em elementos da lista; então, o elemento mínimo é o início da lista, e o elemento máximo é o fim. Se a lista é ***não ordenada***, os elementos podem aparecer em qualquer ordem. Em uma ***lista circular***, o ponteiro *anterior* do início da lista aponta para o fim, e o ponteiro *próximo* do fim da lista aponta para o início. Podemos imaginar uma lista circular

como um anel de elementos. No restante desta seção, supomos que as listas com as quais estamos trabalhando são listas não ordenadas e duplamente ligadas.

Como fazer uma busca em uma lista ligada

O procedimento LIST-SEARCH(L, k) encontra o primeiro elemento com chave k na lista L por meio de uma busca linear simples, retornando um ponteiro para esse elemento. Se nenhum objeto com chave k aparecer na lista, o procedimento retorna NIL. No caso da lista ligada da Figura 10.3(a), a chamada LIST-SEARCH(L, 4) retorna um ponteiro para o terceiro elemento, e a chamada LIST-SEARCH(L, 7) retorna NIL.

Figura 10.3 (**a**) Uma lista duplamente ligada L representando o conjunto dinâmico $\{1, 4, 9, 16\}$. Cada elemento na lista é um objeto com atributos para a chave e ponteiros (mostrados por setas) para o próximo objeto e para o objeto anterior. O atributo *próximo* do fim e o atributo *anterior* do início são NIL, indicados por uma barra diagonal. O atributo *L.início* aponta para o início. (**b**) Seguindo a execução de LIST-INSERT(L, x), onde $x.chave = 25$, a lista ligada tem um novo objeto com chave 25 como o novo início. Esse novo objeto aponta para o antigo início com chave 9.(**c**) O resultado da chamada subsequente LIST-DELETE(L, x), onde x aponta para o objeto com chave 4.

LIST-SEARCH(L, k)

1 $x = L.início$
2 **while** $x \neq$ NIL e $x.chave \neq k$
3 $\quad x = x.próximo$
4 **return** x

Para fazer uma busca em uma lista de n objetos, o procedimento LIST-SEARCH demora o tempo Q(n) no pior caso, já que talvez tenha de pesquisar a lista inteira.

Inserção em uma lista ligada

Dado um elemento x cujo atributo *chave* já foi definido, o procedimento LIST-INSERT "emenda" x à frente da lista ligada, como mostra a Figura 10.3(b).

LIST-INSERT(L, x)

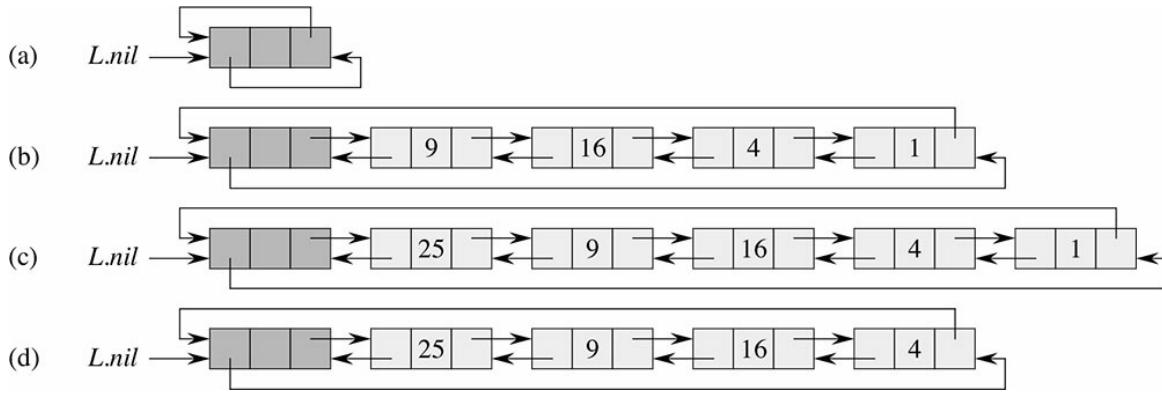
1 $x.próximo = L.início$
2 **if** $L.início \neq$ NIL
3 $\quad L.início.anterior = x$
4 $L.início = x$
5 $x.anterior =$ NIL

(Lembre-se de que nossa notação de atributo pode ser usada em cascata, de modo que *L.início.anterior* denota o atributo *anterior* do objeto que *L.início* aponta.) O tempo de execução para LIST-INSERT para uma lista de n elementos é $O(1)$.

Eliminação em uma lista ligada

O procedimento LIST-DELETE remove um elemento x de uma lista ligada L. Ele deve receber um ponteiro para x, e depois "desligar " x da lista atualizando os ponteiros. Se desejarmos eliminar um elemento com determinada chave, deveremos primeiro chamar LIST-SEARCH, para reaver um ponteiro para o elemento.

```
LIST-DELETE(L, x)
1  if x.anterior ≠ NIL
2     x.anterior.próximo = x.próximo
3  else L.início = x.próximo
4  if x.próximo ≠ NIL
5     x.próximo.anterior = x.anterior
```

A Figura 10.3(c) mostra como um elemento é eliminado de uma lista ligada. LIST-DELETE é executado no tempo $O(1)$ mas, se desejarmos eliminar um elemento com uma dada chave, será necessário o tempo $Q(n)$ no pior caso porque primeiro devemos chamar LIST-SEARCH.

Sentinelas

O código para LIST-DELETE seria mais simples se pudéssemos ignorar as condições de contorno no início e no fim da lista.

```
LIST-DELETE′(L, x)
1  x.anterior.próximo = x.próximo
2  x.próximo.anterior = x.anterior
```

Uma ***sentinela*** é um objeto fictício que nos permite simplificar condições de contorno. Por exemplo, suponha que suprimos com uma lista L um objeto *L.nil* que representa NIL, mas tem todos os atributos dos outros objetos da lista. Onde quer que tenhamos uma referência a NIL no código da lista, nós a substituímos por uma referência à sentinela *L.nil.* Como mostra a Figura 10.4, essa mudança transforma uma lista duplamente ligada normal em uma ***lista circular duplamente ligada com uma sentinela***, na qual a sentinela *L.nil* se encontra entre o início e o fim; o atributo *L.nulo.próximo* aponta para o início da lista e *L.nil.anterior* aponta para o fim. De modo semelhante, tanto o atributo *próximo* do fim quanto o atributo *anterior* do início apontam para *L.nil.* Visto que *L.nil.próximo* aponta para o início, podemos eliminar totalmente o atributo *L.início*, substituindo as referências a ele por referências a *L.nil.próximo.* A Figura 10.4(a) mostra que uma lista vazia consiste apenas na sentinela e que *L.nil.próximo* e *L.nil.anterior* apontam para *L.nil.*

O código para LIST-SEARCH permanece o mesmo de antes, porém com as referências a NIL e *Linício* modificadas como já especificado:

Figura 10.4 Uma lista circular duplamente ligada com uma sentinela. A sentinela *L.nil* aparece entre o início e o fim. O atributo *L.início* não é mais necessário, visto que podemos acessar o início da lista por *L.nil.próximo*. (**a**) Uma lista vazia. (**b**) A lista ligada da Figura 10.3(a), com chave 9 no início e chave 1 no fim. (**c**) A lista após a execução de LIST-INSERT'(L, x), onde $x.chave = 25$. O novo objeto se torna o início da lista. (**d**) A lista após a eliminação do objeto com chave 1. O novo fim é o objeto com chave 4.

```
LIST-SEARCH'(L, k)
1  x = L.nil.próximo
2  while x ≠ L.nil e x.chave ≠ k
3      x = x.próximo
4  return x
```

Usamos o procedimento de duas linhas LIST-DELETE' de antes para eliminar um elemento da lista. O seguinte procedimento insere um elemento na lista:

```
LIST-INSERT'(L, x)
1  x.próximo = L.nil.próximo
2  L.nil.próximo.anterior = x
3  L.nil.próximo = x
4  x.anterior = L.nil
```

A Figura 10.4 mostra os efeitos de LIST-INSERT' e LIST-DELETE' sobre uma amostra de lista.

Sentinelas raramente reduzem os limites assintóticos de tempo de operações de estrutura de dados, mas podem reduzir fatores constantes. O ganho da utilização de sentinelas dentro de laços em geral é uma questão de clareza de código em vez de velocidade; por exemplo, o código da lista ligada fica mais simples quando usamos sentinelas, mas poupamos apenas o tempo $O(1)$ nos procedimentos LIST-INSERT' e LIST-DELETE'. Contudo, em outras situações, a utilização de sentinelas ajuda a restringir o código em um laço, reduzindo assim o coeficiente de, digamos, n ou n_2 no tempo de execução.

Devemos usar sentinelas com sensatez. Quando houver muitas listas pequenas, o armazenamento extra usado por suas sentinelas poderá representar desperdício significativo de memória. Neste livro, só utilizaremos sentinelas quando elas realmente simplificarem o código.

Exercícios

10.2-1 Você pode implementar a operação de conjuntos dinâmicos INSERT em uma lista simplesmente ligada em tempo $O(1)$? E a operação DELETE ?

10.2-2 Implemente uma pilha usando uma lista simplesmente ligada L. As operações PUSH e POP ainda devem demorar o tempo $O(1)$.

10.2-3 Implemente uma fila por meio de uma lista simplesmente ligada L. As operações ENQUEUE e DEQUEUE ainda devem demorar o tempo $O(1)$.

10.2-4 Como está escrita, cada iteração do laço no procedimento LIST-SEARCH' exige dois testes: um para $x \neq L.nil$ e um para $x.chave \neq k$. Mostre como eliminar o teste para $x \neq L.nil$ em cada iteração.

10.2-5 Implemente as operações de dicionário INSERT, DELETE e SEARCH usando listas circulares simplesmente ligadas. Quais são os tempos de execução dos seus procedimentos?

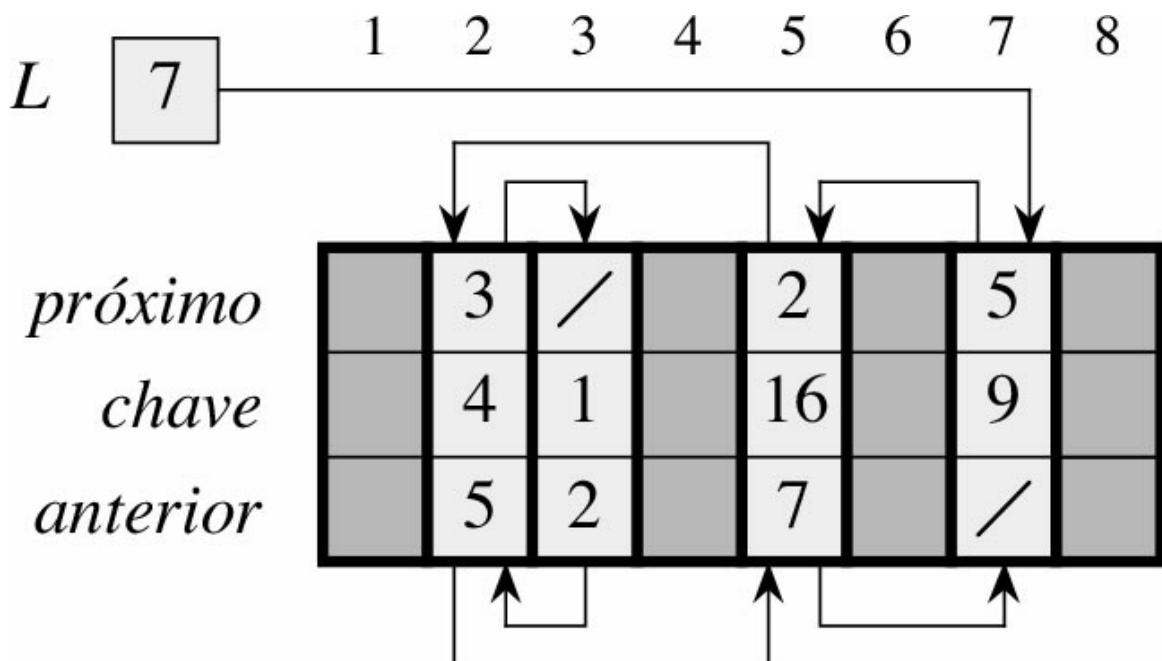
10.2-6 A operação em conjuntos dinâmicos UNION utiliza dois conjuntos disjuntos S_1 e S_2 como entrada e retorna um conjunto $S = S_1 \cup S_2$ que consiste em todos os elementos de S_1 e S_2. Os conjuntos S_1 e S_2 são normalmente destruídos pela operação. Mostre como suportar UNION no tempo $O(1)$ usando uma estrutura de dados de lista adequada.

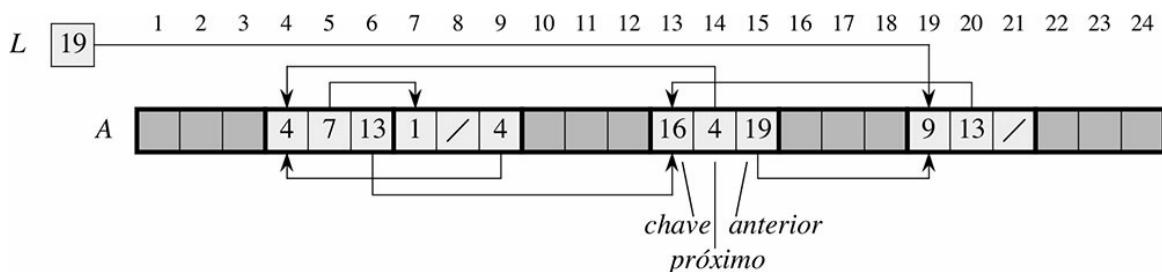
10.2-7 Dê um procedimento não recursivo de tempo $Q(n)$ que inverta uma lista simplesmente ligada de n elementos. O procedimento só pode usar armazenamento constante além do necessário para a própria lista.

10.2-8 ★ Explique como implementar listas duplamente ligadas usando somente um valor de ponteiro $x.np$ por item, em vez dos dois valores usuais (*próximo* e *anterior*). Suponha que todos os valores de ponteiros podem ser interpretados como inteiros de k bits e defina $x.np$ como $x.np = x.próximo$ XOR $x.anterior$ $[x]$ o "ou exclusivo" de k bits de $x.próximo$ e $x.anterior$. (O valor NIL é representado por 0.) Não esqueça de descrever as informações necessárias para acessar o início da lista. Mostre como implementar as operações SEARCH, INSERT e DELETE em tal lista. Mostre também como inverter essa lista em tempo $O(1)$.

10.3 IMPLEMENTAÇÃO DE PONTEIROS E OBJETOS

Como implementamos ponteiros e objetos em linguagens que não os oferecem? Nesta seção, veremos dois modos de implementar estruturas de dados ligadas sem um tipo de dados ponteiro explícito. Sintetizaremos objetos e ponteiros de arranjos e índices de arranjos.

Uma representação de objetos em vários arranjos

Podemos representar uma coleção de objetos que têm os mesmos atributos usando um arranjo para cada atributo. Como exemplo, a Figura 10.5 mostra como podemos implementar a lista ligada da Figura 10.3(a) com três arranjos. A *chave* do arranjo contém os valores das chaves presentes atualmente no conjunto dinâmico, e os ponteiros são armazenados nos arranjos *próximo* e *anterior*. Para um dado índice de arranjo x, $chave[x]$, $próximo[x]$ e $anterior[x]$ representam um objeto na lista ligada. Por essa interpretação, um ponteiro x é simplesmente um índice comum para os arranjos *chave*, *próximo* e *anterior*.

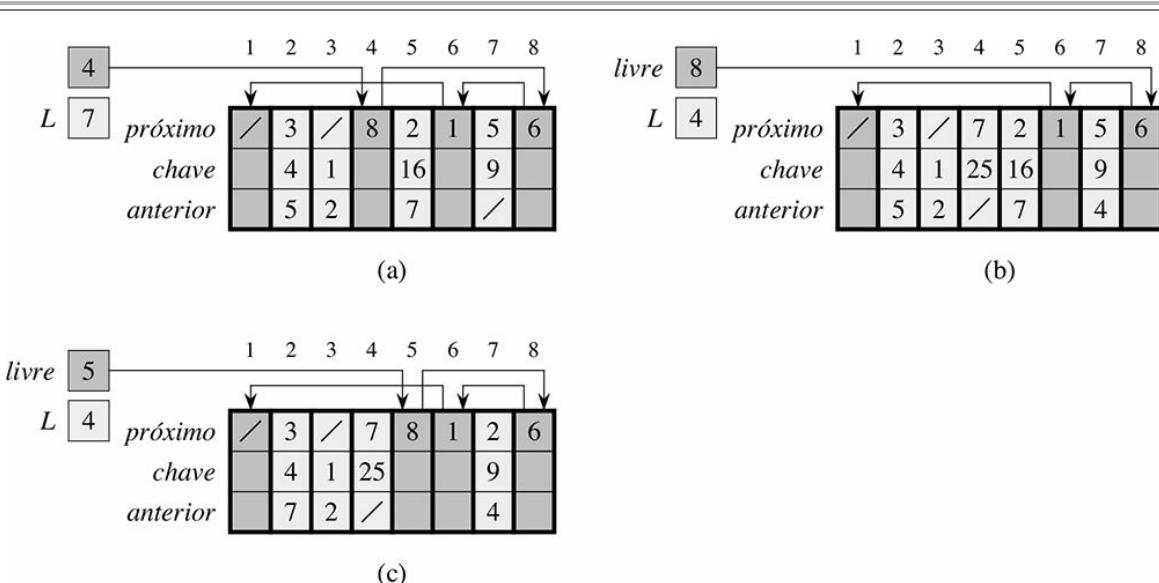
Na Figura 10.3(a), o objeto com chave 4 vem após o objeto com chave 16 na lista ligada. Na Figura 10.5, chave 4 aparece em $chave[2]$ e chave 16 aparece em $chave[5]$; assim, temos $próximo[5] = 2$ e $anterior[2] = 5$. Embora a constante NIL apareça no atributo *próximo* do fim e no atributo *anterior* do início, em geral usamos um inteiro (como 0 ou – 1) que não poderia, de modo algum, representar um índice real para os arranjos. Uma variável L contém o índice do início da lista.

Uma representação de objetos com um único arranjo

As palavras na memória de um computador, normalmente são endereçadas por inteiros de 0 a $M - 1$, onde M é um inteiro adequadamente grande. Em muitas linguagens de programação, um objeto ocupa um conjunto contíguo de posições na memória do computador. Um ponteiro é simplesmente o endereço da primeira posição de memória do objeto, e podemos endereçar outras posições de memória dentro do objeto acrescentando um deslocamento ao ponteiro.

Podemos utilizar a mesma estratégia para implementar objetos em ambientes de programação que não fornecem ponteiros explícitos. Por exemplo, a Figura 10.6 mostra como usar um único arranjo A para armazenar a lista ligada das Figuras 10.3(a) e 10.5. Um objeto ocupa um subarranjo contíguo $A[j .. k]$. Cada atributo do objeto corresponde a um deslocamento na faixa 0 a $k - j$, e um ponteiro para o objeto é o índice j. Na Figura 10.6, os deslocamentos correspondentes a *chave*, *próximo* e *anterior* são 0, 1 e 2, respectivamente. Para ler o valor de *anterior*$[i]$, dado um ponteiro i, adicionamos o valor i do ponteiro ao deslocamento 2, lendo assim $A[i + 2]$.

Figura 10.5 A lista ligada da Figura 10.3(a) representada pelos arranjos *chave*, *próximo* e *anterior*. Cada fatia vertical dos arranjos representa um único objeto. Os ponteiros armazenados correspondem aos índices do arranjo mostrados na parte superior; as setas mostram como interpretá-los. As posições de objetos sombreadas em tom mais claro contêm elementos de listas. A variável L mantém o índice do início.

Figura 10.6 A lista ligada das Figuras 10.3(a) e 10.5, representada em um único arranjo A. Cada elemento da lista é um objeto que ocupa um subarranjo contíguo de comprimento 3 dentro do arranjo. Os três atributos *chave*, *próximo* e *anterior* correspondem aos deslocamentos 0, 1 e 2, respectivamente, dentro de cada objeto. Um ponteiro para um objeto é um índice do primeiro elemento do objeto. Objetos que contêm elementos da lista estão sombreados em tom mais claro, e as setas mostram a ordenação da lista.

A representação de um único arranjo é flexível no sentido de que permite que objetos de diferentes comprimentos sejam armazenados no mesmo arranjo. O problema de administrar tal coleção heterogênea de objetos é mais difícil que o problema de administrar uma coleção homogênea, onde todos os objetos têm os mesmos atributos. Visto que a maioria das estruturas de dados que consideraremos são compostas por elementos homogêneos, será suficiente para nossa finalidade empregar a representação de objetos em vários arranjos.

Alocação e liberação de objetos

Para inserir uma chave em um conjunto dinâmico representado por uma lista duplamente ligada, devemos alocar um ponteiro a um objeto que não está sendo utilizado na representação da lista ligada no momento considerado. Por isso, é útil gerenciar o armazenamento de objetos não utilizados na representação da lista ligada nesse momento, de tal modo que um objeto possa ser alocado. Em alguns sistemas, um ***coletor de lixo*** é responsável por determinar quais objetos não são utilizados. Porém, muitas aplicações são tão simples que podem assumir a responsabilidade pela devolução de um objeto não utilizado a um gerenciador de armazenamento. Agora, exploraremos o problema de alocar e liberar (ou desalocar) objetos homogêneos utilizando o exemplo de uma lista duplamente ligada representada por vários arranjos.

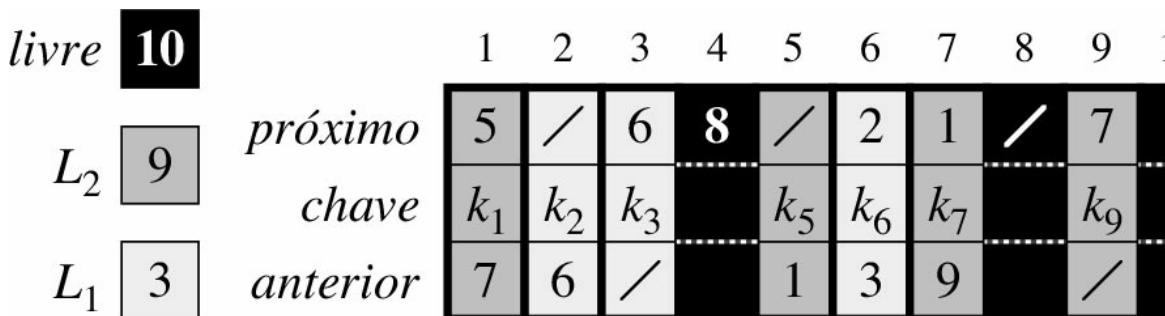
Suponha que os arranjos na representação de vários arranjos tenham comprimento m e que em algum momento o conjunto dinâmico contenha $n \leq m$ elementos. Então, n objetos representam elementos que se encontram atualmente no conjunto dinâmico, e os $m - n$ objetos restantes são ***livres***; os objetos livres estão disponíveis para representar elementos inseridos no conjunto dinâmico no futuro.

Mantemos os objetos livres em uma lista simplesmente ligada, que denominamos ***lista livre***. A lista livre usa apenas o arranjo *próximo*, que armazena os ponteiros *próximo* na lista. O início da lista livre está contido na variável global *livre*. Quando o conjunto dinâmico representado pela lista ligada L é não vazio, a lista livre pode estar entrelaçada com a lista L, como mostra a Figura 10.7. Observe que cada objeto na representação está na lista L ou na lista livre, mas não em ambas.

A lista livre funciona como uma pilha: o próximo objeto alocado é o último objeto liberado.

Figura 10.7 O efeito dos procedimentos ALLOCATE-OBJECT e FREE-OBJECT. (**a**) A lista da Figura 10.5 (sombreada em tom mais claro) e uma lista livre (sombreada em tom mais escuro). As setas mostram a estrutura da lista livre. (**b**) O resultado da chamada ALLOCATE-OBJECT( ) (que retorna o *índice 4*), que define *chave*4 como 25, e chama LIST-INSERT(L, 4). O novo início da lista livre é o objeto 8, que era *próximo*4 na lista livre. (**c**) Após executar LIST-DELETE(L, 5), chamamos FREE-OBJECTS(5). O objeto 5 se torna o novo início da lista livre, seguido pelo objeto 8 na lista livre.

Podemos usar uma implementação de lista das operações de pilhas PUSH e POP, para implementar os procedimentos para alocar e liberar objetos, respectivamente. Consideramos que a variável global *livre* usada nos procedimentos a seguir, aponta para o primeiro elemento da lista livre.

```
ALLOCATE-OBJECT( )
1  if livre == NIL
2      error "out of space"
3  else x = livre
4      livre = x.próximo
5      return x
```

```
FREE-OBJECT(x)
1  x.próximo = livre
2  livre = x
```

A lista livre contém, inicialmente, todos os n objetos não alocados. Assim que a lista livre é esgotada, o procedimento ALLOCATE-OBJECT sinaliza um erro. Podemos até mesmo atender a várias listas ligadas com apenas uma única lista livre. A Figura 10.8 mostra duas listas ligadas e uma lista livre entrelaçadas por meio de arranjos *chave*, *próximo* e *anterior*.

Os dois procedimentos são executados no tempo $O(1)$, o que os torna bastante práticos. Podemos modificá-los de modo que funcionem para qualquer coleção homogênea de objetos permitindo que qualquer um dos atributos no objeto aja como um atributo *próximo* na lista livre.

Figura 10.8 Duas listas ligadas, L_1 (sombreada em tom mais claro) e L_2 (sombreada em tom mais escuro), e uma lista livre (em negro) entrelaçada.

Exercícios

10.3-1 Trace um quadro da sequência ⟨13, 4, 8, 19, 5, 11⟩ armazenada como uma lista duplamente ligada utilizando a representação de vários arranjos. Faça o mesmo para a representação de um único arranjo.

10.3-2 Escreva os procedimentos ALLOCATE-OBJECT e FREE-OBJECT para uma coleção homogênea de objetos implementada pela representação de um único arranjo.

10.3-3 Por que não precisamos definir ou redefinir os atributos *anterior* de objetos na implementação dos procedimentos ALLOCATE-OBJECT e FREE-OBJECT?

10.3-4 Muitas vezes, é desejável manter todos os elementos de uma lista duplamente ligada de forma compacta no armazenamento usando, por exemplo, as primeiras *m* posições do índice na representação de vários arranjos. (Esse é o caso em um ambiente de computação de memória virtual paginada.) Explique como implementar os procedimentos ALLOCATE-OBJECT e FREE-OBJECT de modo que a representação seja compacta. Suponha que não existem ponteiros para elementos da lista ligada fora da própria lista. (*Sugestão*: Use a implementação de uma pilha em arranjo.)

10.3-5 Seja *L* uma lista duplamente ligada de comprimento *n* armazenada em arranjos *chave*, *anterior* e *próximo* de comprimento *m*. Suponha que esses arranjos sejam gerenciados por procedimentos ALLOCATE-OBJECT e FREE-OBJECT que mantêm uma lista livre duplamente ligada *F*. Suponha ainda que, dos *m* itens, exatamente *n* estejam na lista *L* e $m - n$ na lista livre. Escreva um procedimento COMPACTIFY-LIST(*L*, *F*) que, dadas a lista *L* e a lista livre *F*, desloque os itens em *L* de modo que ocupem as posições de arranjo 1, 2, ..., *n* e ajuste a lista livre *F* para que ela permaneça correta, ocupando as posições de arranjo $m + 1$, $m + 2$, ..., *n*. O tempo de execução do seu procedimento deve ser *Q*(*m*), e ele deve utilizar somente uma quantidade constante de espaço extra. Justifique que seu procedimento está correto.

10.4 REPRESENTAÇÃO DE ÁRVORES ENRAIZADAS

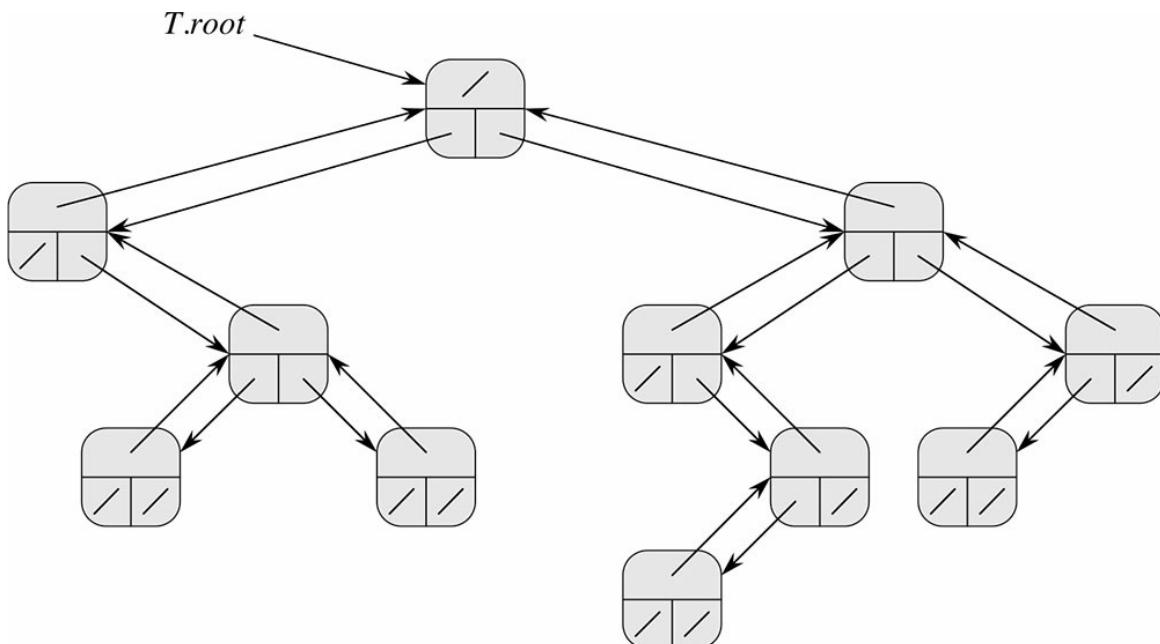
Os métodos para representar listas dados na seção anterior se estendem a qualquer estrutura de dados homogênea. Nesta seção, examinaremos especificamente o problema da representação de árvores enraizadas por estruturas de dados ligadas. Primeiro, veremos as árvores binárias e depois apresentaremos um método para árvores enraizadas nas quais os nós podem ter um número arbitrário de filhos.

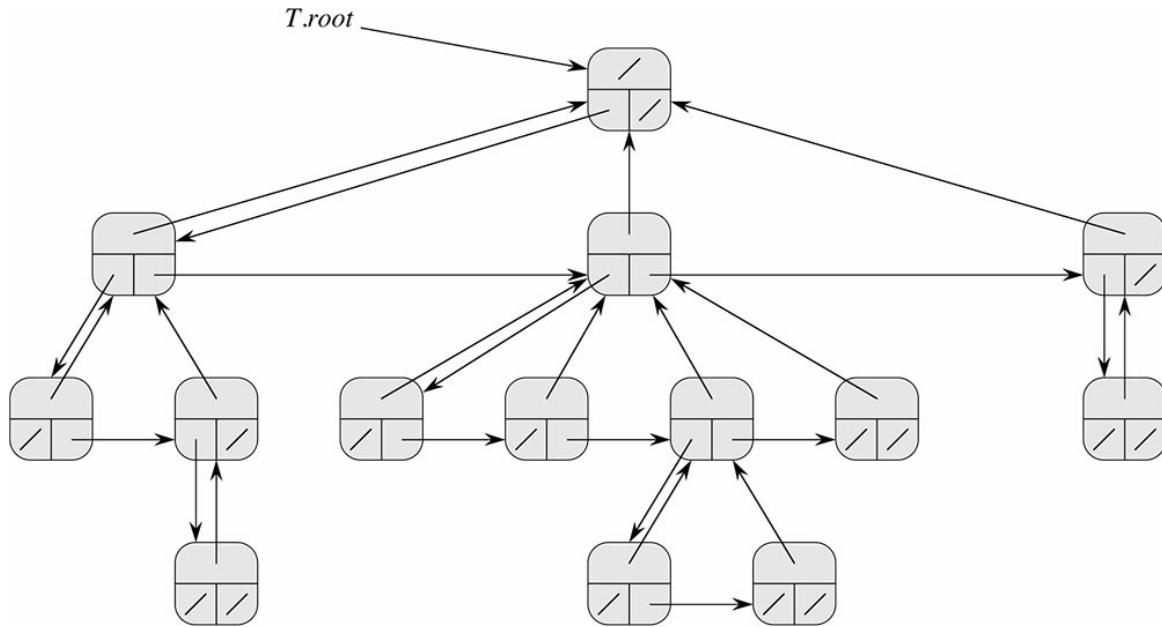
Representamos cada nó de uma árvore por um objeto. Como no caso das listas ligadas, supomos que cada nó contém um atributo *chave*. Os atributos de interesse restantes são ponteiros para outros nós e variam de acordo com o tipo de árvore.

Árvores binárias

A Figura 10.9 mostra como usamos os atributos *p*, *esquerdo* e *direito* para armazenar ponteiros para o pai, o filho da esquerda e o filho da direita de cada nó em uma árvore binária *T*. Se *x*.*p* = NIL, então *x* é a raiz. Se o nó *x* não tem nenhum filho à esquerda, então *x*.*esquerdo* = NIL, e o mesmo ocorre para o filho à direita. A raiz da árvore *T* inteira é apontada pelo atributo *T.raiz*. Se *T.raiz* = NIL, então a árvore é vazia.

Árvores enraizadas com ramificações ilimitadas

Podemos estender o esquema para representar uma árvore binária a qualquer classe de árvores na qual o número de filhos de cada nó seja no máximo alguma constante *k*: substituímos os atributos *esquerdo* e *direito* por $filho_1$, $filho_2$, ..., $filho_k$. Esse esquema deixa de funcionar quando o número de filhos de um nó é ilimitado, já que não sabemos quantos atributos (arranjos na representação de vários arranjos) devemos alocar antecipadamente. Além disso, ainda que o número de filhos *k* seja limitado por uma constante grande, mas a maioria dos nós tenha um número pequeno de filhos, é possível que desperdicemos grande quantidade de memória.

Felizmente, existe um esquema inteligente para representar árvores com números arbitrários de filhos. Tal esquema tem a vantagem de utilizar somente o espaço *O*(*n*) para qualquer árvore enraizada de *n* nós. A ***representação filho da esquerda*, *irmão da direita*** aparece na Figura 10.10. Como antes, cada nó contém um ponteiro pai *p*, e *T.raiz* aponta para a raiz da árvore *T*. Contudo, em vez de ter um ponteiro para cada um de seus filhos, cada nó *x* tem somente dois ponteiros:

1. *x.filho-esquerdo* aponta para o filho da extremidade esquerda do nó *x*, e
2. *x.irmão-direito* aponta para o irmão de *x* imediatamente à sua direita.

Se o nó *x* não tem nenhum filho, então *x.filho-esquerdo* = NIL e, se o nó *x* é o filho da extrema direita de seu pai, então *x.irmão-direito* = NIL.

Outras representações de árvores

Algumas vezes, representamos árvores enraizadas de outras maneiras. Por exemplo, no Capítulo 6, representamos um heap, que é baseado em uma árvore binária completa, por um único arranjo mais o índice do último nó no heap. As árvores que aparecem no Capítulo 21 são percorridas somente em direção à raiz; assim, apenas os ponteiros pais estão presentes: não há ponteiros para filhos. Muitos outros esquemas são possíveis. O melhor esquema dependerá da aplicação.

Figura 10.9 A representação de uma árvore binária *T*. Cada nó *x* tem os atributos *x.p* (superior), *x.esquerdo* (inferior esquerdo) e *x.direito* (inferior direito). Os atributos *chave* não estão mostrados.

Figura 10.10 A representação filho da esquerda, irmão da direita de uma árvore *T*. Cada nó *x* tem atributos *x.p* (superior), *x.filho-esquerdo* (inferior esquerdo) e *x.irmão-direito* (inferior direito). As chaves não são mostradas.

Exercícios

10.4-1 Desenhe a árvore binária enraizada no índice 6 que é representada pelos seguintes atributos:

índice	chave	esquerdo	direito
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2 Escreva um procedimento recursivo de tempo $O(n)$ que, dada uma árvore binária de n nós, imprima a chave de cada nó na árvore.

10.4-3 Escreva um procedimento não recursivo de tempo $O(n)$ que, dada uma árvore binária de n nós, imprima a chave de cada nó na árvore. Use uma pilha como estrutura de dados auxiliar.

10.4-4 Escreva um procedimento de tempo $O(n)$ que imprima todas as chaves de uma árvore enraizada arbitrária com n nós, onde a árvore é armazenada usando a representação filho-esquerdo, irmão-direito.

10.4-5 ★ Escreva um procedimento não recursivo de tempo $O(n)$ que, dada uma árvore binária de n nós, imprima a chave de cada nó. Não utilize mais que um espaço extra constante fora da própria árvore e não modifique a árvore, mesmo temporariamente, durante o procedimento.

10.4-6 ★ A representação filho-esquerdo, irmão-direito de uma árvore enraizada arbitrária utiliza três ponteiros em cada nó: *filho da esquerda*, *irmão da direita* e *pai*. De qualquer nó, seu pai pode ser alcançado e identificado em tempo constante e todos os seus filhos podem ser alcançados e identificados em tempo linear em relação ao número de filhos. Mostre como usar somente dois ponteiros e um valor booleano em cada nó para que o pai de um nó ou todos os seus filhos possam ser alcançados e identificados em tempo linear em relação ao número de filhos.

Problemas

10-1 ***Comparações entre listas***

Para cada um dos quatro tipos de listas na tabela a seguir, qual é o tempo de execução assintótico do pior caso para cada operação em conjuntos dinâmicos apresentada na lista?

	não ordenada, simplesmente ligada	ordenada, simplesmente ligada	não ordenada, duplamente ligada	ordenada, duplamente ligada
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10-2 ***Heaps intercaláveis com a utilização de listas ligadas***

Um heap intercalável suporta as seguintes operações: MAKE-HEAP (que cria um heap intercalável vazio), INSERT, MINIMUM, EXTRACT-MIN e UNION.[1] Mostre como implementar heaps intercaláveis com a utilização de listas ligadas em cada um dos casos a seguir. Procure tornar cada operação tão eficiente quanto possível. Analise o tempo de execução de cada operação em termos do tamanho do(s) conjunto(s) dinâmico(s) sobre o(s) qual(is) é realizada a operação.

a. As listas são ordenadas.

b. As listas são não ordenadas.

c. As listas são não ordenadas e os conjuntos dinâmicos a serem intercalados são disjuntos.

10-3 ***Busca em uma lista compacta ordenada***

O Exercício 10.3-4 perguntou como poderíamos manter uma lista de n elementos compactamente nas primeiras n posições de um arranjo. Suporemos que todas as chaves são distintas e que a lista compacta é também ordenada, isto é, $chave[i] < chave[próximo[i$ para todo $i = 1, 2, ..., n$ tal que $próximo[i] \neq$ NIL. Suporemos também que temos uma variável L que contém o índice do primeiro elemento na lista. Com base nessas premissas, você mostrará que podemos usar o seguinte algoritmo aleatorizado para fazer uma busca na lista no tempo esperado $O(\sqrt{n})$:

```
COMPACT-LIST-SEARCH(L, n, k)
1   i = L
2   while i ≠ NIL and chave[i] < k
3       j = RANDOM (1, n)
4       if chave[i] < chave[j] and chave[j] ≤ i
5           i = j
6           if chave[i] == k
7               return i
8       i = próximo[i]
9   if i == NIL or chave[i] > k
10      return NIL
11  else return i
```

Se ignorarmos as linhas 3–7 do procedimento, teremos um algoritmo comum para busca em uma lista ligada ordenada, no qual o índice i aponta para cada posição da lista por vez. A busca termina assim que o índice i "cair " do final da lista ou assim que $chave[i] \geq k$. Neste último caso, se $chave[i] = k$, fica claro que encontramos uma chave com o valor k. Se, porém, $chave[i] > k$, isso significa que nunca encontraremos uma chave com o valor k. Todavia, se $chave[i] >$ k, então nunca encontraremos uma chave com o valor k e, portanto, encerrar a busca era a coisa certa a fazer.

As linhas 3–7 tentam saltar à frente até uma posição j escolhida aleatoriamente. Esse salto nos beneficiará se $chave[j]$ for maior que $chave[i]$ e não maior que k; nesse caso, j marca uma posição na lista, a qual i teria de alcançar durante uma busca comum na lista. Como a lista é compacta, sabemos que qualquer escolha de j entre 1 e n indexa algum objeto na lista, em vez de uma lacuna na lista livre.

Em vez de analisar o desempenho de COMPACT-LIST-SEARCH diretamente, analisaremos um algoritmo relacionado, COMPACT-LIST-SEARCH', que executa dois laços separados. Esse algoritmo adota um parâmetro adicional t que determina um limite superior para o número de iterações do primeiro laço.

```
COMPACT-LIST-SEARCH'(L, n, k, t)
1   i = L
2   for q = 1 to t
3       j = RANDOM(1, n)
4       if chave[i] < chave[j] e chave[j] ≤ k
5           i = j
6           if chave[i] = k
7               return i
8   while i ≠ NIL e chave[i] < k
9       i = próximo[i]
10  if i = NIL ou chave[i] > k
11      return NIL
12  else return i
```

Para comparar a execução dos algoritmos COMPACT-LIST-SEARCH(L, n, k) e COMPACT-LIST-SEARCH'(L, n, k, t), suponha que a sequência de inteiros retornados pelas chamadas de RANDOM$(1, n)$ é a mesma para ambos os algoritmos.

a. Suponha que COMPACT-LIST-SEARCH(L, n, k) execute t iterações do laço **while** das linhas 2–8. Demonstre que COMPACT-LIST-SEARCH'(L, n, k, t) retorna a mesma resposta e que o número total de iterações de ambos

os laços **for** e **while** dentro de COMPACT-LIST-SEARCH' é pelo menos t.

Na chamada COMPACT-LIST-SEARCH'(L, n, k, t), seja X_t a variável aleatória que descreve a distância na lista ligada (isto é, do começo ao fim da cadeia de ponteiros *próximo*) da posição i até a chave desejada k, após t iterações do laço **for** das linhas 2–7.

b. Mostre que o tempo de execução esperado de COMPACT-LIST-SEARCH'(L, n, k, t) é $O(t + E[X_t])$.

c. Mostre que $E[X_t] \leq \sum_{r=1}^{n}(1 - r/n)^t$. (*Sugestão*: Use a equação (C.25).)

d. Mostre que $\sum_{r=1}^{n} r^t \leq n^{t+1}/(t+1)$.

e. Prove que $EX_t \leq n/(t + 1)$.

f. Mostre que COMPACT-LIST-SEARCH'(L, n, k, t) é executado no tempo esperado $O(t + n/t)$.

g. Conclua que COMPACT-LIST-SEARCH é executado no tempo esperado $O(\sqrt{n})$.

h. Por que supusemos que todas as chaves são distintas em COMPACT-LIST-SEARCH? Demonstre que, saltos aleatórios não necessariamente ajudam assintoticamente quando a lista contém valores de chave repetidos.

NOTAS DO CAPÍTULO

Aho, Hopcroft e Ullman [6] e Knuth [209] são excelentes referências para estruturas de dados elementares. Muitos outros textos focalizam estruturas de dados básicas e também sua implementação em uma linguagem de programação particular. Alguns exemplos desses tipos de livros didáticos são Goodrich e Tamassia [147], Main [241], Shaffer [311] e Weiss [352, 353, 354]. Gonnet [145] fornece dados experimentais sobre o desempenho de muitas operações em estruturas de dados.

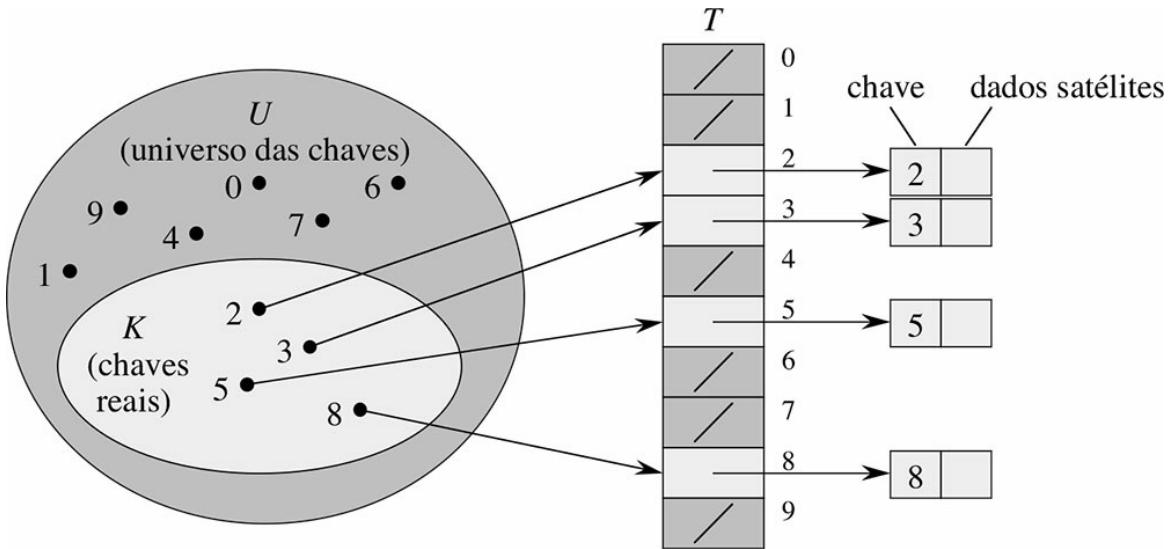
A origem de pilhas e filas como estruturas de dados em ciência da computação não é clara, visto que já existiam noções correspondentes em matemática e em práticas comerciais em papel antes da introdução dos computadores digitais. Knuth [209] cita A. M. Turing sobre o desenvolvimento de pilhas para o encadeamento de sub-rotinas em 1947.

Estruturas de dados baseadas em ponteiros também parecem ser uma invenção folclórica. De acordo com Knuth, ponteiros eram aparentemente usados nos primeiros computadores com memórias de tambor. A linguagem A-1, desenvolvida por G. M. Hopper em 1951, representava fórmulas algébricas como árvores binárias. Knuth credita à linguagem IPL-II, desenvolvida em 1956 por A. Newell, J. C. Shaw e H. A. Simon, o reconhecimento da importância e a promoção do uso de ponteiros. Sua linguagem IPL-III, desenvolvida em 1957, incluía operações explícitas de pilhas.

---

1 Visto que definimos um heap intercalável para suportar Minimum e Extract-Min, também podemos nos referir a ele como um ***heap intercalável de mínimo***. **Alternativamente**, se o heap suportasse Maximun e Extract-Max, ele seria um ***heap intercalável de máximo***.

11 TABELAS DE ESPALHAMENTO

Muitas aplicações exigem um conjunto dinâmico que suporte somente as operações de dicionário INSERT, SEARCH e DELETE. Por exemplo, um compilador que traduz uma linguagem de programação mantém uma tabela de símbolos na qual as chaves de elementos são cadeias de caracteres arbitrários que correspondem a identificadores na linguagem. Uma tabela de espalhamento ou hashing é uma estrutura de dados eficaz para implementar dicionários. Embora a busca por um elemento em uma tabela de espalhamento possa demorar tanto quanto procurar um elemento em uma lista ligada o tempo $\Theta(n)$ no pior caso , na prática o hashing funciona extremamente bem. Sob premissas razoáveis, o tempo médio para pesquisar um elemento em uma tabela de espalhamento é $O(1)$.

Uma tabela de espalhamento generaliza a noção mais simples de um arranjo comum. O endereçamento direto em um arranjo comum faz uso eficiente de nossa habilidade de examinar uma posição arbitrária em um arranjo no tempo $O(1)$. A Seção 11.1 discute o endereçamento direto com mais detalhes. Podemos tirar proveito do endereçamento direto quando temos condições de alocar um arranjo que tem uma única posição para cada chave possível.

Quando o número de chaves realmente armazenadas é pequeno em relação ao número total de chaves possíveis, as tabelas de espalhamento se tornam uma alternativa eficaz para endereçar diretamente um arranjo, já que normalmente uma tabela de espalhamento utiliza um arranjo de tamanho proporcional ao número de chaves realmente armazenadas. Em vez de usar a chave diretamente como um índice de arranjo, o índice de arranjo é *computado* a partir da chave. A Seção 11.2 apresenta as principais ideias, focalizando o "encadeamento" como um modo de tratar "colisões", nas quais mais de uma chave é mapeada para o mesmo índice de arranjo. A Seção 11.3 descreve como podemos computar os índices de arranjos a partir das chaves com o uso de funções hash. Apresentamos e analisamos diversas variações sobre o tema básico. A Seção 11.4 examina o "endereçamento aberto", que é um outro modo de lidar com colisões. A conclusão é que o hash é uma técnica extremamente eficaz e prática: as operações básicas de dicionário exigem apenas o tempo $O(1)$ em média. A Seção 11.5 explica como o "hashing perfeito" pode suportar buscas no tempo do *pior caso* $O(1)$, quando o conjunto de chaves que está sendo armazenado é estático (isto é, quando o conjunto de chaves nunca muda uma vez armazenado).

11.1 TABELAS DE ENDEREÇO DIRETO

O endereçamento direto é uma técnica simples que funciona bem quando o universo U de chaves é razoavelmente pequeno. Suponha que uma aplicação necessite de um conjunto dinâmico no qual cada elemento tem uma chave extraída do universo $U = \{0, 1, ..., m - 1\}$, onde m não é muito grande. Consideramos que não há dois elementos com a mesma chave.

Para representar o conjunto dinâmico, usamos um arranjo, ou uma ***tabela de endereços diretos***, denotada por $T[0 .. m - 1]$, na qual cada ***posição***, ***ou lacuna***, corresponde a uma chave no universo U. A Figura 11.1 ilustra a abordagem; a posição k aponta para um elemento no conjunto com chave k. Se o conjunto não contém nenhum elemento com chave k, então $T[k]$ = NIL.

Figura 11.1 Como implementar um conjunto dinâmico por uma tabela de endereços diretos T. Cada chave no universo $U = \{0, 1, ..., 9\}$ corresponde a um índice na tabela. O conjunto $K = \{2, 3, 5, 8\}$ de chaves reais determina as posições na tabela que contêm ponteiros para elementos. As outras posições, sombreadas em tom mais escuro, contêm NIL.

A implementação das operações de dicionário é trivial.

```
DIRECT-ADDRESS-SEARCH(T, k)
1    return T[k] = x

DIRECT-ADDRESS-INSERT(T, x)
1    T[x.chave] = x

DIRECT-ADDRESS-DELETE(T, x)
1    T[x.chave] = NIL
```

Cada uma dessas operações leva somente o tempo $O(1)$.

Em algumas aplicações, a própria tabela de endereços diretos pode conter os elementos do conjunto dinâmico. Isto é, em vez de armazenar a chave e os dados satélites de um elemento em um objeto externo à tabela de endereços diretos, com um ponteiro de uma posição na tabela até o objeto, podemos armazenar o objeto na própria posição e, portanto, economizar espaço. Usaríamos uma chave especial dentro de um objeto para indicar uma lacuna. Além disso, muitas vezes, é desnecessário armazenar a chave do objeto, já que, se temos o índice de um objeto na tabela, temos sua chave. Entretanto, se as chaves não forem armazenadas, devemos ter algum modo de saber se a posição está vazia.

Exercícios

11.1-1 Suponha que um conjunto dinâmico S seja representado por uma tabela de endereços diretos T de comprimento m. Descreva um procedimento que determine o elemento máximo de S. Qual é o desempenho do pior caso do seu procedimento?

11.1-2 *Um **vetor de bits*** é simplesmente um arranjo de bits (0s e 1s). Um vetor de bits de comprimento m ocupa um espaço muito menor que um arranjo de m ponteiros. Descreva como usar um vetor de bits para representar um conjunto dinâmico de elementos distintos sem dados satélites. Operações de dicionário devem ser executadas no tempo $O(1)$.

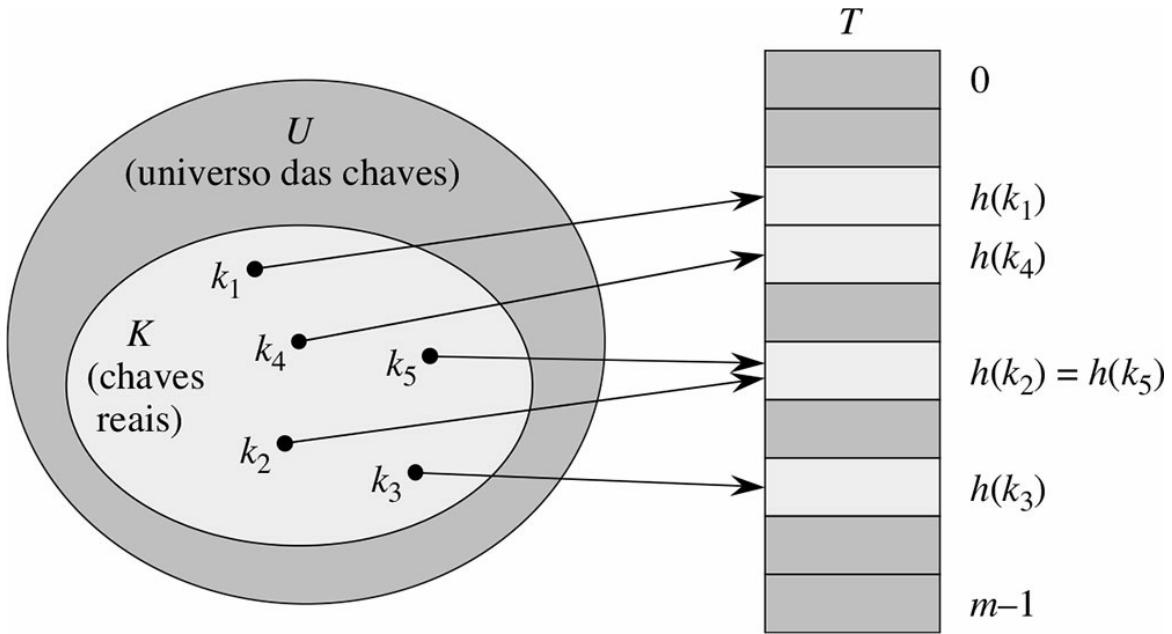
11.1-3 Sugira como implementar uma tabela de endereços diretos na qual as chaves de elementos armazenados não precisem ser distintas e os elementos possam ter dados satélites. Todas as três operações de dicionário (INSERT, DELETE e SEARCH) devem ser executadas no tempo $O(1)$. (Não esqueça que DELETE adota como argumento um ponteiro para um objeto a ser eliminado, não uma chave.)

11.1-4 ★ Desejamos implementar um dicionário usando endereçamento direto para um arranjo *enorme*. No início, as entradas do arranjo podem conter lixo, e inicializar o arranjo inteiro é impraticável devido ao seu tamanho. Descreva um esquema para implementar um dicionário de endereço direto para um arranjo enorme. Cada objeto armazenado deve utilizar espaço $O(1)$; as operações SEARCH, INSERT e DELETE devem demorar tempo $O(1)$ cada uma e a inicialização da estrutura de dados deve demorar o tempo $O(1)$. (*Sugestão*: Use um arranjo adicional tratado de certo modo como uma pilha cujo tamanho é o número de chaves realmente armazenadas no dicionário, para ajudar a determinar se uma dada entrada no arranjo enorme é válida ou não.)

11.2 TABELAS DE ESPALHAMENTO

O aspecto negativo do endereçamento direto é óbvio: se o universo U é grande, armazenar uma tabela T de tamanho $|U|$ pode ser impraticável ou mesmo impossível, dada a memória disponível em um computador típico. Além disso, o conjunto K de chaves *realmente armazenadas* pode ser tão pequeno em relação a U que grande parte do espaço alocado para T seria desperdiçada.

Quando o conjunto K de chaves armazenadas em um dicionário é muito menor que o universo U de todas as chaves possíveis, uma tabela de espalhamento requer armazenamento muito menor que uma tabela de endereços diretos. Especificamente, podemos reduzir o requisito de armazenamento a $(|K|)$ e ao mesmo tempo manter o benefício de procurar um elemento na tabela ainda no tempo $O(1)$. A pegada é que esse limite é para o *caso do tempo médio*, enquanto no caso do endereçamento direto ele vale para o *tempo do pior caso*.

Com endereçamento direto, um elemento com a chave k é armazenado na posição k. Com hash, esse elemento é armazenado na posição $h(k)$; isto é, usamos uma ***função hash*** h para calcular a posição pela chave k. Aqui, h mapeia o universo U de chaves para as posições de uma ***tabela de espalhamento*** $T[0 \mathinner{..} m-1]$:

$$h : U \rightarrow \{0, 1, ..., m-1\} ,$$

onde o tamanho m da tabela de espalhamento normalmente é muito menor que $|U|$. Dizemos que um elemento com a chave k ***se espalha*** (hashes) até a posição $h(k)$; dizemos também que $h(k)$ é o ***valor hash*** da chave k. A Figura 11.2 ilustra a ideia básica. A função hash reduz a faixa de índices do arranjo e, consequentemente, o tamanho do arranjo. Em vez de ter tamanho $|U|$, o arranjo pode ter tamanho m.

Porém há um revés: após o hash, duas chaves podem ser mapeadas para a mesma posição. Chamamos essa situação de ***colisão***. Felizmente, existem técnicas eficazes para resolver o conflito criado por colisões.

É claro que a solução ideal seria evitar por completo as colisões. Poderíamos tentar alcançar essa meta escolhendo uma função hash adequada h. Uma ideia é fazer h parecer "aleatória", evitando assim as colisões ou ao menos minimizando seu número. A expressão "to hash", em inglês, que evoca imagens de misturas e retalhamentos aleatórios, capta o espírito dessa abordagem. (É claro que uma função hash h deve ser determinística, no sentido de que uma dada entrada k sempre deve produzir a mesma saída $h(k)$.) Porém, como $|U| > m$, devem existir no mínimo duas chaves que têm o mesmo valor hash; portanto, evitar totalmente as colisões é impossível. Assim, embora uma função hash bem projetada e de aparência "aleatória" possa minimizar o número de colisões, ainda precisamos de um método para resolver as colisões que ocorrerem.

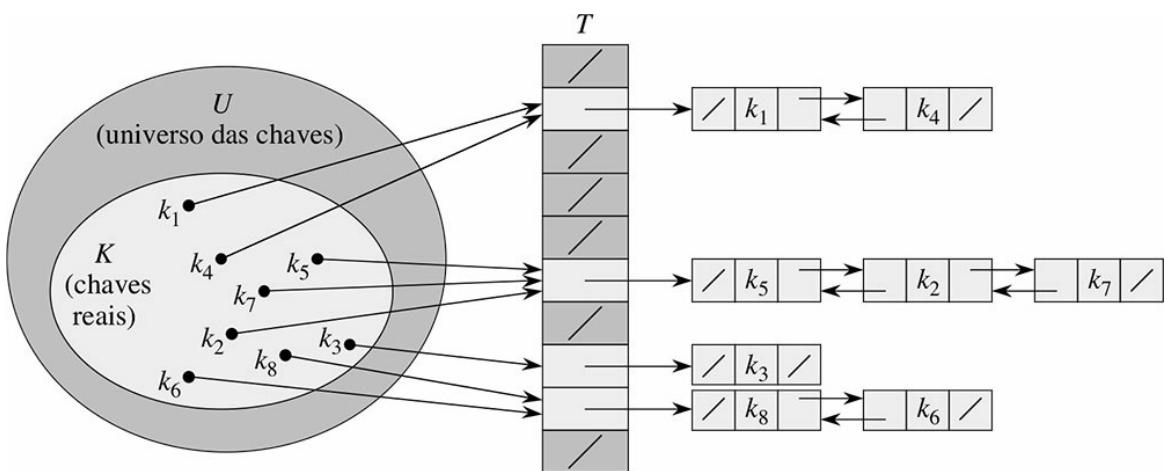
O restante desta seção apresenta a técnica mais simples para resolução de colisões, denominada encadeamento. A Seção 11.4 apresenta um método alternativo para resolver colisões, denominado endereçamento aberto.

Figura 11.2 Utilização de uma função hash h para mapear chaves para posições de uma tabela de espalhamento. Como são mapeadas para a mesma posição, as chaves k_2 e k_5 colidem.

Resolução de colisões por encadeamento

No ***encadeamento***, todos os elementos resultantes do hash vão para a mesma posição em uma lista ligada, como mostra a Figura 11.3. A posição j contém um ponteiro para o início da lista de todos os elementos armazenados que, após o hash, foram para j; se não houver nenhum desses elementos, a posição j contém NIL.

As operações de dicionário em uma tabela de espalhamento T são fáceis de implementar quando as colisões são resolvidas por encadeamento.

CHAINED-HASH-INSERT(T, x)
1 insere x no início da lista $T[h(x.chave)]$

CHAINED-HASH-SEARCH(T, k)
1 procura um elemento com a chave k na lista $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
1 elimina x da lista $T[h(x.chave)]$

Figura 11.3 Resolução de colisão por encadeamento. Cada posição Tj da tabela de espalhamento contém uma lista ligada de todas as chaves cujo valor hash é j. Por exemplo, $h(k_1) = h(k_4)$ e $h(k_5) = h(k_2) = h(k_7)$. A lista ligada pode ser simplesmente ligada ou duplamente ligada; aqui ela é mostrada como duplamente ligada porque assim a eliminação é mais rápida.

O tempo de execução do pior caso para inserção é $O(1)$. O procedimento de inserção é rápido em parte porque supõe que o elemento x que está sendo inserido ainda não está presente na tabela ; se necessário, podemos verificar essa premissa (a custo adicional) procurando um elemento cuja chave é $x.chave$ antes da inserção. Para busca, o tempo de execução do pior caso é proporcional ao comprimento da lista; analisaremos essa operação mais atentamente em seguida. Podemos eliminar um elemento x no tempo $O(1)$ se as listas forem duplamente ligadas, como mostra a Figura 11.3. (Observe que CHAINED-HASH-DELETE toma como entrada um elemento x e não sua chave k, de modo que não temos de procurar x antes. Se a tabela de espalhamento suportar eliminação, então suas listas ligadas devem ser duplamente ligadas para podermos eliminar um item rapidamente. Se as listas forem apenas simplesmente ligadas, para eliminar o elemento x, em primeiro lugar teríamos de encontrar x na lista $T[h(x.chave)]$ para podermos atualizar o próximo atributo do predecessor de x. Com listas simplesmente ligadas, a eliminação e a busca teriam os mesmos tempos de execução assintóticos.)

Análise do hash com encadeamento

Como é o desempenho do hashing com encadeamento? Em particular, quanto tempo ele leva para procurar um elemento com uma determinada chave?

Dada uma tabela de espalhamento T com m posições que armazena n elementos, definimos o ***fator de carga*** α para T como n/m, isto é, o número médio de elementos armazenados em uma cadeia. Nossa análise será em termos de α, que pode ser menor, igual ou maior que 1.

O comportamento do pior caso do hashing com encadeamento é terrível: todas as n chaves vão para a mesma posição após o hashing, criando uma lista de comprimento n. Portanto, o tempo do pior caso para a busca é (n) mais o tempo necessário para calcular a função hash — não é melhor do que seria se usássemos uma única lista ligada para todos os elementos. É claro que não usamos as tabelas de espalhamento por seu desempenho no pior caso. (Todavia, o hashing perfeito, descrito na Seção 11.5, realmente oferece bom desempenho no pior caso quando o conjunto de chaves é estático.)

O desempenho do hashing para o caso médio depende de como a função hash h distribui o conjunto de chaves a armazenar entre as m posições, em média. A Seção 11.3 discute essas questões, mas, por enquanto, devemos considerar que qualquer elemento dado tem igual probabilidade de passar para qualquer das m posições após o hashing, independentemente do lugar para onde qualquer outro elemento tenha passado após essa operação. Denominamos essa premissa ***hashing uniforme simples***.

Para $j = 0, 1, ..., m - 1$, vamos denotar o comprimento da lista $T[j]$ por n_j, de modo que

$$n = n_0 + n_1 + ..., + n_{m-1}, \tag{11.1}$$

e o valor esperado de n_j é $E[n_j] = \alpha = n/m$.

Supomos que o tempo $O(1)$ é suficiente para calcular o valor hash $h(k)$, de modo que o tempo requerido para procurar um elemento com chave k depende linearmente do comprimento $n_{h(k)}$ da lista $T[h(k)]$. Deixando de lado o tempo $O(1)$ necessário para calcular a função hash e acessar a posição $h(k)$, vamos considerar o número esperado de elementos examinados pelo algoritmo de busca, isto é, o número de elementos na lista $T[h(k)]$ que o algoritmo verifica para ver se qualquer deles tem uma chave igual a k. Consideraremos dois casos. No primeiro, a busca não é bem-sucedida: nenhum elemento na tabela tem a chave k. No segundo caso, a busca consegue encontrar um elemento com chave k.

Teorema 11.1

Em uma tabela de espalhamento na qual as colisões são resolvidas por encadeamento, uma busca mal sucedida demora o tempo do caso médio $(1 + \alpha)$, sob a hipótese de hashing uniforme simples.

Prova Sob a hipótese de hashing uniforme simples, qualquer chave k ainda não armazenada na tabela tem igual probabilidade de ocupar qualquer das m posições após o hash. O tempo esperado para procurar sem sucesso uma chave k é o tempo esperado para pesquisar até o fim da lista $T[h(k)]$, que tem o comprimento esperado $E[n_{h(k)}] = \alpha$. Assim, o número esperado de elementos examinados em uma busca mal sucedida é α, e o tempo total exigido (incluindo o tempo para se calcular $h(k)$) é $(1 + \alpha)$.

A situação para uma busca bem-sucedida é ligeiramente diferente, já que a probabilidade de cada lista ser pesquisada não é a mesma. Em vez disso, a probabilidade de uma lista ser pesquisada é proporcional ao número de elementos que ela contém. Todavia, o tempo de busca esperado ainda é $(1 + \alpha)$.

Teorema 11.2

Em uma tabela de espalhamento na qual as colisões são resolvidas por encadeamento, uma busca bem-sucedida demora o tempo do caso médio $(1 + \alpha)$ se considerarmos hashing uniforme simples.

Prova Supomos que o elemento que está sendo pesquisado tem igual probabilidade de ser qualquer dos n elementos armazenados na tabela. O número de elementos examinados durante uma busca bem-sucedida para um elemento x é uma unidade maior que o número de elementos que aparecem antes de x na lista de x. Como elementos novos são colocados à frente na lista, os elementos antes de x na lista foram todos inseridos após x ser inserido. Para determinar o número esperado de elementos examinados, tomamos a média, sobre os n elementos x na tabela, de 1 mais o número esperado de elementos adicionados à lista de x depois que x foi adicionado à lista. Seja x_i o i-ésimo elemento inserido na tabela, para $i = 1, 2, ..., n$, e seja $k_i = x_i.chave$. Para chaves k_i e k_j, definimos a variável aleatória indicadora $X_{ij} = I\{h(k_i) = h(k_j)\}$. Sob a hipótese de hashing uniforme simples, temos $\Pr\{h(k_i) = h(k_j)\} = 1/m$ e, então, pelo Lema 5.1, $E[X_{ij}] = 1/m$. Assim, o número esperado de elementos examinados em uma busca bem-sucedida é

$$
\begin{aligned}
& E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} E[X_{ij}]\right) \quad \text{por linearidade da esperança} \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} \frac{1}{m}\right) \\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n} n-\sum_{i=1}^{n} i\right) \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \quad \text{pela equação (A.1))} \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}
$$

Assim, o tempo total exigido para uma busca bem-sucedida (incluindo o tempo para calcular a função hash) é $(2 + \alpha/2 - \alpha/2n) = (1 + \alpha)$.

O que significa essa análise? Se o número de posições da tabela de espalhamento é no mínimo proporcional ao número de elementos na tabela, temos $n = O(m)$ e, consequentemente, $\alpha = n/m = O(m)/m = O(1)$. Assim, a busca

demora tempo constante na média. Visto que a inserção demora o tempo $O(1)$ no pior caso e a eliminação demora o tempo $O(1)$ no pior caso quando as listas são duplamente ligadas, podemos suportar todas as operações de dicionário no tempo $O(1)$ na média.

Exercícios

11.2-1 Suponha que utilizamos uma função hash h para efetuar o hashing de n chaves distintas em um arranjo T de comprimento m. Considerando hashing uniforme simples, qual é o número esperado de colisões? Mais precisamente, qual é a cardinalidade esperada de $\{\{k, l\} : k \neq l \text{ e } h(k) = h(l)\}$?

11.2-2 Demonstre o que acontece quando inserimos as chaves 5, 28, 19, 15, 20, 33, 12, 17, 10 em uma tabela de espalhamento com colisões resolvidas por encadeamento. Considere uma tabela com nove posições e a função hash $h(k) = k \bmod 9$.

11.2-3 O professor Marley apresenta a hipótese de que podemos obter ganhos substanciais de desempenho modificando o esquema de encadeamento para manter cada lista em sequência ordenada. Como a modificação do professor afeta o tempo de execução para pesquisas bem-sucedidas, mal sucedidas, inserções e eliminações?

11.2-4 Sugira como alocar e desalocar armazenamento para elementos dentro da própria tabela de espalhamento ligando todas as posições não utilizadas em uma lista livre. Suponha que uma posição pode armazenar um sinalizador e um elemento mais um ponteiro ou dois ponteiros. Todas as operações de dicionário e de lista livre devem ser executadas no tempo esperado $O(1)$. A lista livre precisa ser duplamente ligada ou uma lista livre simplesmente ligada é suficiente?

11.2-5 Suponha que estejamos armazenando um conjunto de n chaves em uma tabela de espalhamento de tamanho m. Mostre que, se as chaves forem extraídas de um universo U com $|U| > nm$, então U tem um subconjunto de tamanho n composto por chaves que passam todas para uma mesma posição após o hashing, de modo que o tempo de busca do pior caso para o hashing com encadeamento é (n).

11.2-6 Suponha que armazenemos n chaves em uma tabela de espalhamento de tamanho m, com colisões resolvidas por encadeamento e que conhecemos o comprimento de cada cadeia, incluindo o comprimento L da cadeia mais longa. Descreva um procedimento que seleciona uma chave uniformemente ao acaso entre as chaves na tabela de espalhamento e a retorna no tempo esperado $O(L \cdot (1 + 1/\alpha))$.

11.3 Funções hash

Nesta seção, discutiremos algumas questões relacionadas ao projeto de boas funções hash e depois apresentaremos três esquemas para sua criação. Dois dos esquemas, hash por divisão e hash por multiplicação, são heurísticos por natureza, enquanto o terceiro esquema, hash universal, utiliza a aleatorização para oferecer um desempenho que podemos provar que é bom.

O que faz uma boa função hash ?

Uma boa função hash satisfaz (aproximadamente) a premissa do hashing uniforme simples: cada chave tem igual probabilidade de passar para qualquer das m posições por uma operação de hash, independentemente da posição que qualquer outra chave ocupou após o hash. Infelizmente, normalmente não temos nenhum meio de verificar essa condição, já que raramente conhecemos a distribuição de probabilidade da qual as chaves são extraídas. Além disso, as

chaves poderiam não ser extraídas independentemente. Ocasionalmente conhecemos a distribuição. Por exemplo, se soubermos que as chaves são números reais aleatórios k, independente e uniformemente distribuídos na faixa $0 \le k < 1$, então a função hash

$$h(k) = km$$

satisfaz a condição de hashing uniforme simples.

Na prática, muitas vezes, podemos usar técnicas heurísticas para criar uma função hash que funciona bem. Informações qualitativas sobre a distribuição de chaves podem ser úteis nesse processo de projeto. Por exemplo, considere a tabela de símbolos de um compilador, na qual as chaves são cadeias de caracteres que representam identificadores em um programa. Símbolos estreitamente relacionados, como pt e pts, frequentemente ocorrem no mesmo programa. Uma boa função hash minimizaria a chance de tais variações passarem para a mesma posição após o hashing.

Uma boa abordagem deriva o valor hash de um modo que esperamos seja independente de quaisquer padrões que possam existir nos dados. Por exemplo, o "método de divisão" (discutido na Seção 11.3.1) calcula o valor hash como o resto quando a chave é dividida por um número primo especificado. Esse método frequentemente dá bons resultados, desde que que escolhamos um número primo que não esteja relacionado com quaisquer padrões na distribuição de chaves.

Finalmente, observamos que algumas aplicações de funções hash poderiam exigir propriedades mais fortes que as oferecidas pelo hashing uniforme simples. Por exemplo, poderíamos querer que chaves que de certo modo são mais "próximas" derivem valores hash muito afastados um do outro. (Essa propriedade é especialmente desejável quando estamos usando sondagem linear, definida na Seção 11.4.) O hash universal, descrito na Seção 11.3.3, muitas vezes, fornece as propriedades desejadas.

Interpretação de chaves como números naturais

A maior parte das funções hash considera como universo de chaves o conjunto = {0, 1, 2, ...} de números naturais. Assim, se as chaves não forem números naturais, temos de encontrar um modo de *interpretá-las* como números naturais. Por exemplo, podemos interpretar uma cadeia de caracteres como um inteiro expresso em uma notação de raiz adequada. Assim, poderíamos interpretar o identificador pt como o par de inteiros decimais (112, 116), já que p = 112 e t = 116 no conjunto de caracteres ASCII; então, expresso como um inteiro de raiz 128, pt se torna $(112 \cdot 128) + 116 = 14.452$. No contexto de uma determinada aplicação, normalmente podemos elaborar algum método para interpretar cada chave como um número natural (possivelmente grande). No que vem a seguir, supomos que as chaves são números naturais.

11.3.1 O MÉTODO DE DIVISÃO

No ***método de divisão*** para criar funções hash, mapeamos uma chave k para uma de m posições, tomando o resto da divisão de k por m. Isto é, a função hash é

$$h(k) = k \bmod m .$$

Por exemplo, se a tabela de espalhamento tem tamanho $m = 12$ e a chave é $k = 100$, então $h(k) = 4$. Visto que exige uma única operação de divisão, o hash por divisão é bastante rápido.

Quando utilizamos o método de divisão, em geral evitamos certos valores de m. Por exemplo, m não deve ser uma potência de 2, já que, se $m = 2^p$, então $h(k)$ será somente o grupo de p bits de ordem mais baixa de k. A menos que saibamos que todos os padrões de p bits de ordem baixa são igualmente prováveis, é melhor garantir que em nosso projeto a função hash dependa de todos os bits da chave. Como o Exercício 11.3-3 lhe pede para mostrar, escolher $m = 2^p - 1$ quando k é uma cadeia de caracteres interpretada em raiz 2^p pode ser uma escolha ruim porque permutar os caracteres de k não altera seu valor hash.

Muitas vezes, um primo não muito próximo de uma potência exata de 2 é uma boa escolha para m. Por exemplo, suponha que desejemos alocar uma tabela de espalhamento, com colisões resolvidas por encadeamento, para conter aproximadamente $n = 2.000$ cadeias de caracteres, onde um caractere tem oito bits. Não nos importamos de examinar uma média de três elementos em uma busca mal sucedida e, assim, alocamos uma tabela de espalhamento de tamanho $m >= 701$. Pudemos escolher o número 701 porque é um primo próximo de 2.000/3, mas não próximo de nenhuma potência de 2. Tratando cada chave k como um inteiro, nossa função hash seria

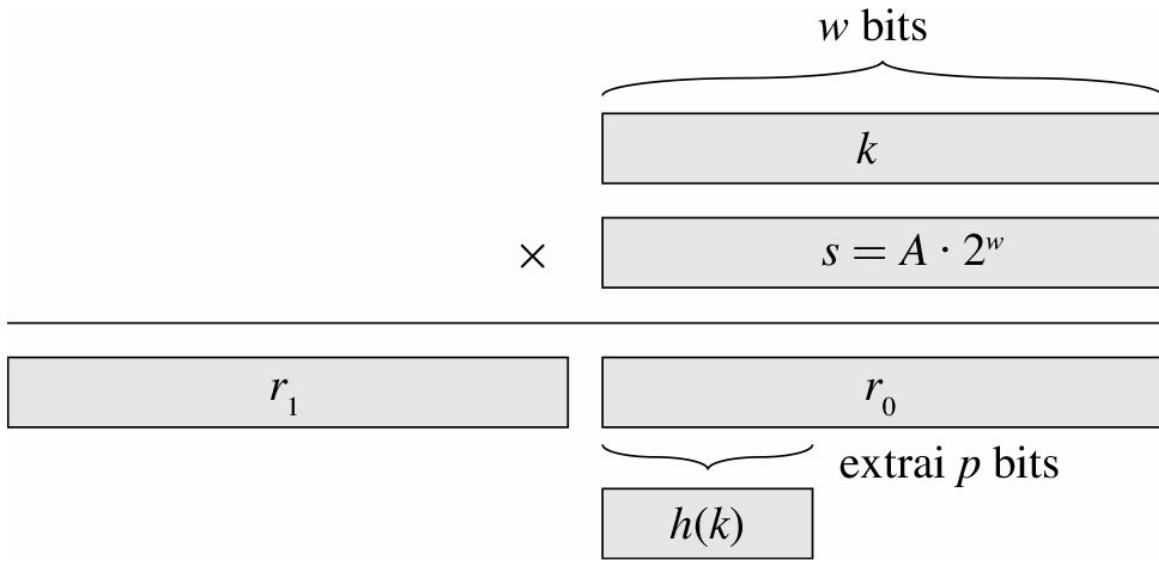
$$h(k) = k \bmod 701 .$$

11.3.2 O MÉTODO DE MULTIPLICAÇÃO

O ***método de multiplicação*** para criar funções hash funciona em duas etapas. Primeiro, multiplicamos a chave k por uma constante A na faixa $0 < A < 1$ e extraímos a parte fracionária de kA. Em seguida, multiplicamos esse valor por m e tomamos o piso do resultado. Resumindo, a função hash é

$$h(k) = \lfloor m(k\,A \bmod 1) \rfloor \ ,$$

onde "$k\,A$ mod 1" significa a parte fracionária de kA, isto é, $kA - kA$.

Uma vantagem do método de multiplicação é que o valor de m não é crítico. Em geral, nós o escolhemos de modo a ser uma potência de 2 ($m = 2^p$ para algum inteiro p) já que podemos implementar facilmente a função na maioria dos computadores do modo descrito a seguir. Suponha que o tamanho da palavra da máquina seja w bits e que k caiba em uma única palavra. Restringimos A a ser uma fração da forma $s/2^w$, onde s é um inteiro na faixa $0 < s < 2^w$. Referindo-nos à Figura 11.4, primeiro multiplicamos k pelo inteiro $s = A \cdot 2^w$ de w bits. O resultado é um valor de $2w$ bits $r_1 2^w + r_0$, onde r_1 é a palavra de ordem alta do produto e r_0 é a palavra de ordem baixa do produto. O valor hash de p bits desejado consiste nos p bits mais significativos de r_0.

Embora esse método funcione com qualquer valor da constante A, funciona melhor com alguns valores que com outros. A escolha ótima depende das características dos dados aos quais o hash está sendo aplicado. Knuth [185] sugere que

$$A \approx (\sqrt{5} - 1)/2 = 0{,}6180339887... \qquad (11.2)$$

tem boa possibilidade de funcionar razoavelmente bem.

Como exemplo, suponha que tenhamos $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ e $w = 32$. Adaptando a sugestão de Knuth, escolhemos A como a fração da forma $s/2^{32}$ mais próxima a $(\sqrt{5}-1)/2$, isto é, $A = 2654435769/2^{32}$. Então, $k \times s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$ e, assim, $r_1 = 76300$ e $r_0 = 17612864$. Os 14 bits mais significativos de r_0 formam o valor $h(k) = 67$.

Figura 11.4 O método de multiplicação para hash. A representação de w bits da chave k é multiplicada pelo valor de w bits $s = A \times 2^w$, onde $0 < A < 1$ é uma constante adequada. Os p bits de ordem mais alta da metade inferior de w bits do produto formam o valor hash desejado $h(k)$.

11.3.3 ★ HASHING UNIVERSAL

Se um adversário malicioso escolher as chaves às quais o hashing deverá ser aplicado por alguma função hash fixa, ele pode escolherá n chaves que passem para a mesma posição após o hash, o que resulta em um tempo médio de recuperação igual a Q(n). Qualquer função hash fixa é vulnerável a esse terrível comportamento do pior caso; a única maneira eficaz de melhorar a situação é escolher a função hash *aleatoriamente*, de um modo que seja *independente* das chaves que realmente serão armazenadas. Essa abordagem, denominada ***hashing universal***, pode resultar em um desempenho demonstravelmente bom na média, não importando as chaves escolhidas pelo adversário.

No hashing universal, no início da execução selecionamos a função hash aleatoriamente de uma classe de funções cuidadosamente projetada. Como no caso do quicksort, a aleatorização garante que nenhuma entrada isolada evocará sempre o comportamento do pior caso. Como selecionamos a função hash aleatoriamente, o algoritmo poderá se comportar de modo diferente em cada execução ainda que a entrada seja a mesma, garantindo um bom desempenho do caso médio para qualquer entrada. Retornando ao exemplo da tabela de símbolos de um compilador, verificamos que agora a escolha de identificadores pelo programador não pode provocar consistentemente um desempenho ruim do hash. O desempenho ruim ocorre apenas quando o compilador escolhe uma função hash aleatória que faz com que o resultado do hash aplicado ao conjunto de identificadores seja ruim, mas a probabilidade de ocorrer essa situação é pequena, e é a mesma para qualquer conjunto de identificadores do mesmo tamanho.

Seja uma coleção finita de funções hash que mapeiam um dado universo U de chaves para a faixa $\{0, 1, ..., m - 1\}$. Dizemos que ela é ***universal*** se, para cada par de chaves distintas $k, l \in U$, o número de funções hash $h \in$ para as quais $h(k) = h(l)$ é no máximo $||/m$. Em outras palavras, com uma função hash escolhida aleatoriamente de , a chance de uma colisão entre chaves distintas k e l não é maior que a chance $1/m$ de uma colisão se $h(k)$ e $h(l)$ fossem escolhidas aleatória e independentemente do conjunto $\{0, 1, ..., m - 1\}$.

O teorema a seguir, mostra que uma classe universal de funções hash oferece bom comportamento do caso médio. Lembre-se de que n_i denota o comprimento da lista $T[i]$.

Teorema 11.3

Suponha que uma função hash h seja escolhida aleatoriamente de uma coleção universal de funções hash e usada para aplicar hash às n chaves em uma tabela T de tamanho m, usando encadeamento para resolver colisões. Se a chave k

não estiver na tabela, o comprimento esperado $E[n_{h}(k)]$ da lista para a qual a chave k passa após aplicação do hash é no máximo o fator de carga $\alpha = n/m$. Se a chave k estiver na tabela, o comprimento esperado $E[n_{h}(k)]$ da lista que contém a chave k é no máximo $1 + \alpha$.

Prova Notamos que, aqui, as esperanças referem-se à escolha da função hash e não dependem de quaisquer premissas adotadas para a distribuição das chaves. Para cada par k e l de chaves distintas, defina a variável aleatória indicadora $X_{kl} = I\{h(k) = h(l)\}$. Visto que, pela definição de uma coleção de funções hash, um único par de chaves colide com probabilidade de no máximo $1/m$, temos $\Pr\{h(k) = h(l)\} \leq 1/m$. Portanto, pelo Lema 5.1, temos $[X_{kl}] \leq 1/m$.

Em seguida, definimos para cada chave k a variável aleatória Y_k, que é igual ao número de chaves diferentes de k que passam para a mesma posição de k após a aplicação do hash, de modo que

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl} .$$

Portanto, temos

$$\begin{aligned} E[Y_k] &= E\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\ &= \sum_{\substack{l \in T \\ l \neq k}} E[X_{kl}] \quad \text{(por linearidade da esperança)} \\ &\leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m} \end{aligned}$$

O restante da prova depende de a chave k estar ou não na tabela T.

- Se $k \notin T$, então $n_{h(k)} = Y_k$ e $|\{l : l \in T \text{ e } l \neq k\}| = n$. Assim, $E n_{h(k)} = E Y_k \leq n/m = \alpha$.
- Se $k \in T$, então, como a chave k aparece na lista $Th(k)$ e a contagem Y_k não inclui a chave k, temos $n_{h(k)} = Y_k + 1$ e $|\{l : l \in T \text{ e } l \neq k\}| = n - 1$. Assim, $E[n_{h}(k)] = E[Y_k] \leq (n - 1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$.

O corolário a seguir, diz que o hash universal nos dá a compensação desejada: agora ficou impossível um adversário escolher uma sequência de operações que force o tempo de execução do pior caso. Com a aleatorização inteligente da escolha da função hash em tempo de execução, garantimos que podemos processar toda sequência de operações com um bom tempo de execução do caso médio.

Corolário 11.4

Usando hash universal e resolução de colisão por encadeamento em uma tabela inicialmente vazia com m posições, tratar qualquer sequência de n operações INSERT, SEARCH e DELETE contendo $O(m)$ operações INSERT demora o tempo esperado (n).

Prova Como o número de inserções é $O(m)$, temos $n = O(m)$ e, assim, $\alpha = O(1)$. As operações INSERT e DELETE demoram tempo constante e, pelo Teorema 11.3, o tempo esperado para cada operação SEARCH é $O(1)$. Assim, por linearidade da esperança, o tempo esperado para a sequência de operações inteira é $O(n)$. Visto que cada operação leva o tempo (1), decorre o limite (n).

Projeto de uma classe universal de funções hash

É bastante fácil projetar uma classe universal de funções hash, como um pouco de teoria dos números nos ajudará a demonstrar. Se você não estiver familiarizado com a teoria dos números, será bom consultar o Capítulo 31 antes.

Começamos escolhendo um número primo p suficientemente grande para que toda chave k possível esteja no intervalo 0 a $p - 1$, inclusive. Seja $_p$ o conjunto $\{0, 1, ..., p - 1\}$ e seja $_p$ o conjunto $\{1, 2, ..., p - 1\}$. Visto que p é primo, podemos resolver equações de módulo p com os métodos dados no Capítulo 31. Como supomos que o tamanho do universo de chaves é maior que o número de posições na tabela de espalhamento, temos $p > m$.

Agora definimos a função hash h_{ab} para qualquer $a \in {}_p$ e qualquer $b \in {}_p$ usando uma transformação linear seguida por reduções de módulo p e, então, de módulo m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m \,. \qquad (11.3)$$

Por exemplo, com $p = 17$ e $m = 6$, temos $h_{3,4}(8) = 5$. A família de todas essas funções hash é

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ e } b \in \mathbb{Z}_p.\} \,. \qquad (11.4)$$

Cada função hash $h_{a,b}$ mapeia $_p$ para $_m$. Essa classe de funções hash tem a interessante propriedade de que o tamanho m da faixa de saída é arbitrário não necessariamente primo , uma característica que usaremos na Seção 11.5. Visto que temos $p - 1$ escolhas para a e que há p escolhas para b, a coleção $_{pm}$ contém $p(p - 1)$ funções hash.

Teorema 11.5

A classe $_{pm}$ de funções hash definida pelas equações (11.3) e (11.4) é universal.

Prova Considere duas chaves distintas k e l de $_p$, de modo que $k \neq l$. Para uma dada função hash h_{ab} fazemos

$$r = (ak + b) \bmod p \,,$$

$$s = (al + b) \bmod p \,.$$

Primeiro observamos que $r \neq s$. Por quê? Observe que

$$r - s \equiv a(k - l) \pmod p \,.$$

Decorre que $r \neq s$ porque p é primo e ambos, a e $(k - l)$, não são zero módulo p e, assim, seu produto também deve ser não zero módulo p pelo Teorema 31.6. Portanto, na computação de qualquer h_{ab} em $_{pm}$, entradas distintas k e l mapeiam para valores distintos r e s módulo p; ainda não há nenhuma colisão no "nível mod p". Além disso, cada uma das $p(p - 1)$ escolhas possíveis para o par (a, b) com $a \neq 0$ produz um par resultante (r, s) *diferente* com $r \neq s$, já que podemos resolver para a e b dados r e s:

$$a = ((r - s)((k - l)^{-1} \bmod p)) \bmod p \,,$$

$$b = (r - ak) \bmod p \,,$$

onde $((k - l)^{-1} \bmod p)$ denota o inverso multiplicativo único, módulo p, de $k - l$. Como existem apenas $p(p - 1)$ pares (r, s) possíveis com $r \neq s$, há uma correspondência de um para um entre pares (a, b) com $a \neq 0$ e pares (r, s) com $r \neq s$. Assim, para qualquer par de entradas k e l dado, se escolhermos (a, b) uniformemente ao acaso de $^* \times$, o par resultante (r, s) terá igual probabilidade de ser qualquer par de valores distintos módulo p.

Então, a probabilidade de chaves distintas k e l colidirem é igual à probabilidade de $r \equiv s \pmod m$ quando r e s são escolhidos aleatoriamente como valores distintos módulo p. Para um dado valor de r, dos $p - 1$ valores restantes possíveis para s, o número de valores s tais que $s \neq r$ e $s \equiv r$ é no máximo

$$\lceil p/m \rceil - 1 \leq ((p + m - 1)/m) - 1 \qquad \text{(pela desigualdade (3.6))}$$
$$= (p - 1)/m \,.$$

A probabilidade de s colidir com r quando reduzido módulo m é no máximo $((p - 1)/m)/(p - 1) = 1/m$.

Assim, para qualquer par de valores distintos $k, l \in {}_p$,

$$\Pr\{h_{ab}(k) = h_{ab}(l)\} \leq 1/m \,,$$

de modo que ${}_{pm}$ é, de fato, universal.

Exercícios

11.3-1 Suponha que desejemos buscar em uma lista ligada de comprimento n, onde cada elemento contém um chave k juntamente com um valor hash $h(k)$. Cada chave é uma cadeia de caracteres longa. Como poderíamos tirar proveito dos valores hash ao procurar na lista um elemento com uma chave específica?

11.3-2 Suponha que aplicamos hash a uma cadeia de r caracteres em m posições, tratando-a como um número de raiz 128, depois usando o método de divisão. Podemos representar facilmente o número m como uma palavra de computador de 32 bits, mas a cadeia de r caracteres, tratada como um número de raiz 128, ocupa muitas palavras. Como podemos aplicar o método de divisão para calcular o valor hash da cadeia de caracteres sem usar mais do que um número constante de palavras de armazenamento fora da própria cadeia?

11.3-3 Considere uma versão do método de divisão, na qual $h(k) = k \bmod m$, onde $m = 2^p - 1$ e k é uma cadeia de caracteres interpretada em raiz 2^p. Mostre que, se pudermos derivar a cadeia x da cadeia y por permutação de seus caracteres, o hash aplicado a x e y resultará no mesmo valor para ambos. Dê exemplo de uma aplicação na qual essa propriedade seria indesejável em uma função hash.

11.3-4 Considere uma tabela de espalhamento de tamanho $m = 1.000$ e uma função hash correspondente $h(k)$ igual a $m\ (k\ A \bmod 1)$ para $A = (\sqrt{5}-1)/2$. Calcule as localizações para as quais são mapeadas as chaves 61, 62, 63, 64 e 65.

11.3-5 ★ Defina uma família de funções hash de um conjunto finito U para um conjunto finito B como e-***universal*** se, para todos os pares de elementos distintos k e l em U,

$$\Pr\{h(k) = h(l)\} \leq \epsilon,$$

onde a probabilidade refere-se à escolha aleatória da função hash h na família H. Mostre que uma família e-universal de funções hash deve ter

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{U}$$

11.3-6 ★ Seja U o conjunto de ênuplas de valores extraídos de ${}_p$, e seja $B = {}_p$, onde p é primo. Defina a função hash $h_b : U \rightarrow B$ para $b \in {}_p$ para um ênupla de entrada $\langle a_0, a_1, ..., a_{n-1} \rangle$ extraída de U por

$$h_b(\langle a_0, a_1, \ldots, a_{n-1} \rangle) = \left(\sum_{j=0}^{n-1} a_j b^j \right) \bmod p,$$

e seja $= \{h_b : b \in {}_p.\}$. Demonstre que é $((n-1)/p)$-universal, de acordo com a definição de $\in$-universal no Exercício 11.3-5. (*Sugestão*: Veja o Exercício 31.4-4.)

11.4 ENDEREÇAMENTO ABERTO

Em ***endereçamento aberto***, todos os elementos ficam na própria tabela de espalhamento. Isto é, cada entrada da tabela contém um elemento do conjunto dinâmico ou NIL. Ao procurar um elemento, examinamos sistematicamente as posições da tabela até encontrar o elemento desejado ou até confirmar que o elemento não está na tabela. Diferentemente do encadeamento, não existe nenhuma lista e nenhum elemento armazenado fora da tabela. Assim, no endereçamento aberto, a tabela de espalhamento pode "ficar cheia", de tal forma que nenhuma inserção adicional pode ser feita; o fator de carga α nunca pode exceder 1.

É claro que poderíamos armazenar as listas ligadas para encadeamento no interior da tabela de espalhamento, nas posições não utilizadas de outro modo (veja o Exercício 11.2-4), mas a vantagem do endereçamento aberto é que ele evita por completo os ponteiros. Em vez de seguir ponteiros, *calculamos* a sequência de posições a examinar. A memória extra liberada por não armazenarmos ponteiros fornece à tabela de espalhamento um número maior de posições para a mesma quantidade de memória, o que produz potencialmente menor número de colisões e recuperação mais rápida.

Para executar inserção usando endereçamento aberto, examinamos sucessivamente, ou ***sondamos***, a tabela de espalhamento até encontrar uma posição vazia na qual inserir a chave. Em vez de ser fixa na ordem 0, 1, ..., $m - 1$ (o que exige o tempo de busca $\Theta(n)$), a sequência de posições sondadas *depende da chave que está sendo inserida*. Para determinar quais serão as posições a sondar, estendemos a função hash para incluir o número da sondagem (a partir de 0) como uma segunda entrada. Assim, a função hash se torna

$$h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, \ldots, m-1\} .$$

Com endereçamento aberto, exigimos que, para toda chave k, a ***sequência de sondagem***

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$$

seja uma permutação de $\langle 0, 1, ..., m-1\rangle$, de modo que toda posição da tabela de espalhamento seja eventualmente considerada uma posição para uma nova chave, à medida que a tabela é preenchida. No pseudocódigo a seguir, supomos que os elementos na tabela de espalhamento T são chaves sem informações satélites; a chave k é idêntica ao elemento que contém a chave k. Cada posição contém uma chave ou NIL (se a posição estiver vazia). O procedimento HASH-SEARCH tem como entrada uma tabela de espalhamento T e uma chave k. Devolve o número da posição onde armazena chave k ou sinaliza um erro porque a tabela de espalhamento já está cheia.

```
HASH-INSERT(T, k)
1  i = 0
2  repeat j = h(k, i)
3       if T[j] == NIL
4           T[j] = k
5           return j
6       else i = i + 1
7  until i == m
8  error "estouro da tabela"
```

O algoritmo que procura a chave k sonda a mesma sequência de posições que o algoritmo de inserção examinou quando a chave k foi inserida. Portanto, a busca pode terminar (sem sucesso) quando encontra uma posição vazia, já que k teria sido inserido ali e não mais adiante em sua sequência de sondagem. (Esse argumento supõe que não há eliminação de chaves na tabela de espalhamento.) O procedimento HASH-SEARCH tem como entrada uma tabela de espalhamento T e um chave k, e devolve j se verificar que a posição j contém a chave k, ou NIL se a chave k não estiver presente na tabela T.

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3     j = h(k, i)
4        if T[j] == k
5           return j
6        i = i + 1
7  until T[j] == NIL ou i == m
8  return NIL
```

Eliminar algo em uma tabela de espalhamento de endereço aberto é difícil. Quando eliminamos uma chave da posição i, não podemos simplesmente marcar essa posição como vazia, nela armazenando NIL. Se fizéssemos isso, poderíamos não conseguir recuperar nenhuma chave k em cuja inserção tivéssemos sondado a posição i e verificado que ela estava ocupada. Podemos resolver esse problema marcando a posição nela armazenando o valor especial DELETED em vez de NIL. Então, modificaríamos o procedimento HASH-INSERT para tratar tal posição como se ela estivesse vazia, de modo a podermos inserir uma nova chave. Não é necessário modificar HASH-SEARCH, já que ele passará por valores DELETED enquanto estiver pesquisando. Entretanto, quando usamos o valor especial DELETED, os tempos de busca não dependem mais do fator de carga α. Por essa razão, o encadeamento é mais comumente selecionado como uma técnica de resolução de colisões quando precisamos eliminar chaves.

Em nossa análise, consideramos ***hash uniforme***: a sequência de sondagem de cada chave tem igual probabilidade de ser qualquer uma das $m!$ permutações de $\langle 0, 1, ..., m-1 \rangle$. O hash uniforme generaliza a noção de hash uniforme simples definida anteriormente para uma função hash que produz não apenas um número único, mas toda uma sequência de sondagem. Contudo, o verdadeiro hash uniforme é difícil de implementar e, na prática, são usadas aproximações adequadas (como o hash duplo, definido a seguir).

Examinaremos três técnicas comumente utilizadas para calcular as sequências de sondagem exigidas para o endereçamento aberto: sondagem linear, sondagem quadrática e hash duplo. Todas essas técnicas garantem que $\langle h(k, 0), h(k, 1), ..., h(k, m-1) \rangle$ é uma permutação de $\langle 0, 1, ..., m-1 \rangle$ para cada chave k. Porém, nenhuma delas cumpre o requisito do hash uniforme, já que nenhuma consegue gerar mais de m_2 sequências de sondagem diferentes (em vez das $m!$ que o hash uniforme exige). O hash duplo tem o maior número de sequências de sondagem e, como seria de esperar, parece dar os melhores resultados.

Sondagem linear

Dada uma função hash comum $h'\colon U \rightarrow \{0, 1, ..., m-1\}$, à qual nos referimos como uma ***função hash auxiliar***, o método de ***sondagem linear*** usa a função hash

$$h(k,i) = (h'(k) + i) \bmod m$$

para $i = 0, 1, ..., m-1$. Dada a chave k, primeiro sondamos $T[h'(k)]$, isto é, a posição dada pela função hash auxiliar. Em seguida, sondamos a posição $T[h'(k) + 1]$, e assim por diante até a posição $T[m-1]$. Depois, voltamos às

posições $T[0]$, $T[1]$, ..., até finalmente sondarmos a posição $T[h'(k) - 1]$. Como a sondagem inicial determina toda a sequência de sondagem, há somente m sequências de sondagem distintas.

A sondagem linear é fácil de implementar, mas sofre de um problema conhecido como ***agrupamento primário***. Longas sequências de posições ocupadas se acumulam, aumentando o tempo médio de busca. Os agrupamentos surgem porque uma posição vazia precedida por i posições cheias é preenchida em seguida com probabilidade $(i + 1)/m$. Longas sequências de posições ocupadas tendem a ficar mais longas, e o tempo médio de busca aumenta.

Sondagem quadrática

A ***sondagem quadrática*** utiliza uma função hash da forma

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \,, \qquad (11.5)$$

onde h' é uma função hash auxiliar, c_1 e c_2 são constantes positivas auxiliares e $i = 0, 1, ..., m - 1$. A posição inicial sondada é $T[h'(k)]$; posições posteriores sondadas são deslocadas por quantidades que dependem de modo quadrático do número da sondagem i. Esse método funciona muito melhor que a sondagem linear mas, para fazer pleno uso da tabela de espalhamento, os valores de c_1, c_2 e m são restritos. O Problema 11-3 mostra um modo de selecionar esses parâmetros. Além disso, se duas chaves têm a mesma posição inicial de sondagem, então suas sequências de sondagem são iguais, já que $h(k_1, 0) = h(k_2, 0)$ implica $h(k_1, i) = h(k_2, i)$. Essa propriedade conduz a uma forma mais branda de agrupamento, denominada ***agrupamento secundário***. Como na sondagem linear, a sondagem inicial determina a sequência inteira; assim, apenas m sequências de sondagem distintas são utilizadas.

Hash duplo

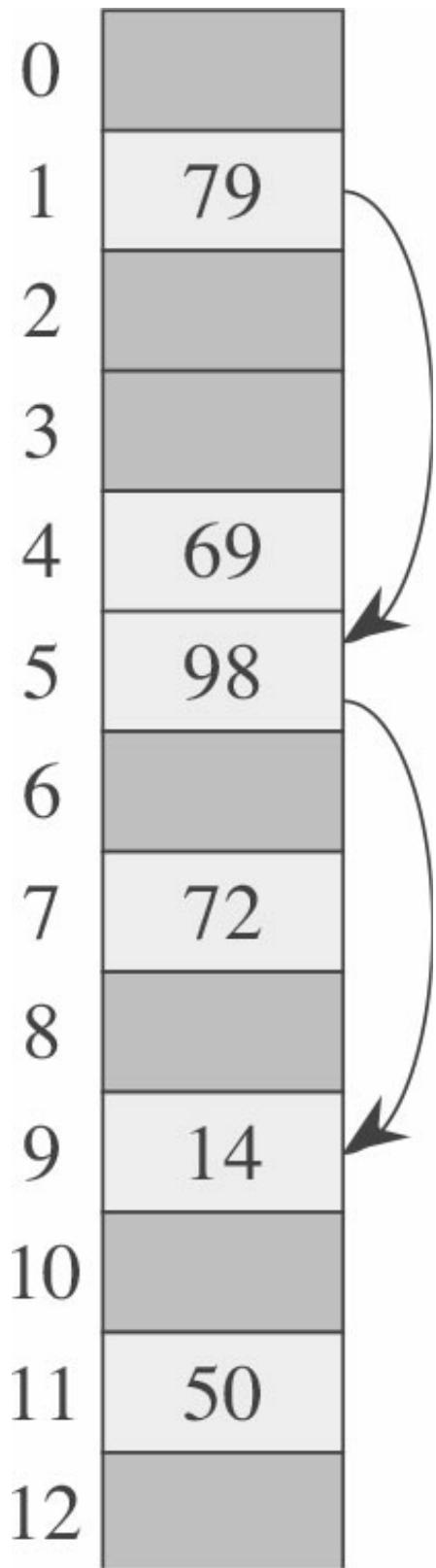
O hash duplo oferece um dos melhores métodos disponíveis para endereçamento aberto porque as permutações produzidas têm muitas das características de permutações escolhidas aleatoriamente. O ***hash duplo*** usa uma função hash da forma

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m \,,$$

onde h_1 e h_2 são funções hash auxiliares. A sondagem inicial vai à posição $T[h_1(k)]$; posições de sondagem sucessivas são deslocadas em relação às posições anteriores pela quantidade $h_2(k)$, módulo m. Assim, diferentemente do caso de sondagem linear ou quadrática, aqui a sequência de sondagem depende da chave k de duas maneiras, já que a posição inicial de sondagem, o deslocamento, ou ambos podem variar. A Figura 11.5 dá um exemplo de inserção por hash duplo.

O valor $h_2(k)$ e o tamanho m da tabela de espalhamento devem ser primos entre si para que a tabela de espalhamento inteira seja examinada (veja o Exercício 11.4-4). Uma forma conveniente de assegurar essa condição é que m seja uma potência de 2 e projetar h_2 de modo que sempre retorne um número ímpar. Outra maneira é que m seja primo e projetar h_2 de modo que sempre retorne um inteiro positivo menor que m. Por exemplo, poderíamos escolher m primo e fazer

$$h_1(k) = k \bmod m \,,$$
$$h_2(k) = 1 + (k \bmod m') \,,$$

onde o valor de m escolhido é ligeiramente menor que m (digamos, $m - 1$). Por exemplo, se $k = 123456$, $m = 701$ e $m = 700$, temos $h_1(k) = 80$ e $h_2(k) = 257$; assim, primeiro sondamos a posição 80 e depois examinamos cada 257ª posição (módulo m) até encontrarmos a chave ou termos examinado todas as posições.

Quando m é primo ou uma potência de 2, o hash duplo é melhor do que a sondagem linear ou quadrática no sentido de que são usadas (m_2) sequências de sondagem em vez de (m), já que cada par $(h_1(k), h_2(k))$ possível gera uma sequência de sondagem distinta. O resultado é que, para tais valores de m, o desempenho do hash duplo parece ficar bem próximo do esquema "ideal" do hash uniforme.

Embora, em princípio, outros valores de m que não sejam primos nem potências de 2 poderiam ser utilizados com hash duplo, na prática torna-se mais difícil gerar $h_2(k)$ eficientemente de um modo que garanta que esse valor e m são primos entre si, em parte porque a densidade relativa $(m)/m$ de tais números pode ser pequena (veja equação (31.24)).

Análise de hash de endereço aberto

Como fizemos na análise do encadeamento, expressamos nossa análise do endereçamento aberto em termos do fator de carga $\alpha = n/m$ da tabela de espalhamento. É claro que no endereçamento aberto temos no máximo um elemento por posição e, portanto, $n \leq m$, o que implica $\alpha \leq 1$.

Figura 11.5 Inserção por hash duplo. Aqui temos uma tabela de espalhamento de tamanho 13 com $h_1(k) = k \bmod 13$ e $h_2(k) = 1 + (k \bmod 11)$. Como $14 \equiv 1 \pmod{13}$ e $14 \equiv 3 \pmod{1}$, inserimos a chave 14 na posição vazia 9, após examinar as posições 1 e 5 verificarmos que elas já estão ocupadas.

Supomos que estamos usando hash uniforme. Nesse esquema idealizado, a sequência de sondagem $\langle h(k, 0), h(k, 1), ..., h(k, m-1)\rangle$ usada para inserir ou procurar cada chave k tem igual probabilidade de ser qualquer permutação de

$\langle 0, 1, ..., m - 1\rangle$. É claro que uma determinada chave tem uma sequência de sondagem fixa e única associada a ela; o que queremos dizer é que, considerando a distribuição de probabilidades no espaço de chaves e a operação da função hash nas chaves, cada sequência de sondagem possível é igualmente provável.

Agora, analisamos o número esperado de sondagens para hash com endereçamento aberto, adotando a premissa do hashing uniforme, começando com uma análise do número de sondagens realizadas em uma busca mal sucedida.

Teorema 11.6

Dada uma tabela de espalhamento de endereço aberto com fator de carga $\alpha = n/m < 1$, o número esperado de sondagens em uma busca mal sucedida é no máximo $1/(1 - \alpha)$, supondo hashing uniforme.

Prova Em uma busca mal sucedida, toda sondagem exceto a última acessa uma posição ocupada que não contém a chave desejada, e a última posição sondada está vazia. Vamos definir a variável aleatória X como o número de sondagens executadas em uma busca mal sucedida, e também definir o evento A_i, para $i = 1, 2, ...$, como o evento em que ocorre uma i-ésima sondagem e ela é para uma posição ocupada. Então, o evento $\{X \geq i\}$ é a interseção dos eventos $A_1 \cap A_2 \cap ... \cap A_{i-1}$. Limitaremos $\{X \geq i\}$ limitando $\Pr\{A_1 \cap A_2 \cap ... \cap A_{i-1}\}$. Pelo Exercício C.2-5,

$$\Pr\{A_1 \cap A_2 \cap ... \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} ... \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap ... \cap A_{i-2}\}.$$

Visto que há n elementos e m posições, $\Pr\{A_1\} = n/m$. Para $j > 1$, a probabilidade de existir uma j-ésima sondagem e ela ser para uma posição ocupada, dado que as primeiras $j - 1$ sondagens foram para posições ocupadas, é $(n - j + 1)/(m - j + 1)$. Essa probabilidade decorre porque estaríamos encontrando um dos $(n - (j - 1))$ elementos restantes em uma das $(m - (j - 1))$ posições não examinadas e, pela premissa do hash uniforme, a probabilidade é a razão entre essas quantidades. Observando que $n < m$ implica $(n - j)/(m - j) \leq n/m$ para todo j tal que $0 \leq j < m$, temos para todo i tal que $1 \leq i \leq m$,

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1}.\end{aligned}$$

Agora, usamos a equação (C.25) para limitar o número esperado de sondagens:

$$\begin{aligned}E[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha}\end{aligned}$$

Esse limite de $1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$ tem uma interpretação intuitiva. Sempre executamos a primeira sondagem. Com probabilidade de aproximadamente α, essa primeira sondagem encontra uma posição ocupada, de modo que precisamos de uma segunda sondagem. Com probabilidade de aproximadamente α^2, as duas primeiras posições estão ocupadas, de modo que executamos uma terceira sondagem, e assim por diante.

Se α é uma constante, o Teorema 11.6 prevê que uma busca mal sucedida é executada no tempo $O(1)$. Por exemplo, se a tabela de espalhamento estiver cheia até a metade, o número médio de sondagens em uma busca mal sucedida é no máximo $1/(1 - 0{,}5) = 2$. Se ela estiver 90% cheia, o número médio de sondagens será no máximo $1/(1 - 0{,}9) = 10$.

O Teorema 11.6 nos dá o desempenho do procedimento HASH-INSERT quase imediatamente.

Corolário 11.7

Inserir um elemento em uma tabela de espalhamento de endereço aberto com fator de carga α exige no máximo $1/(1 - \alpha)$ sondagens, em média, considerando hash uniforme.

Prova Um elemento é inserido somente se houver espaço na tabela e, portanto, $\alpha < 1$. Inserir uma chave requer uma busca mal sucedida seguida pela colocação da chave na primeira posição vazia encontrada. Assim, o número esperado de sondagens é, no máximo, $1/(1 - \alpha)$.

Temos de trabalhar um pouco mais para calcular o número esperado de sondagens para uma busca bem-sucedida.

Teorema 11.8

Dada uma tabela de espalhamento de endereço aberto com fator de carga $\alpha < 1$, o número esperado de sondagens em uma busca bem sucedida é, no máximo,
considerando hash uniforme e também que cada chave na tabela tem igual probabilidade de ser procurada.

Prova Uma busca de uma chave k reproduz a mesma sequência de sondagem que foi seguida na inserção do elemento com chave k. Pelo Corolário 11.7, se k foi a $(i + 1)$-ésima chave inserida na tabela de espalhamento, o número esperado de sondagens efetuadas em uma busca de k é no máximo $1/(1 - i/m) = m/(m - i)$. O cálculo da média para todas as n chaves na tabela de espalhamento nos dá o número esperado de sondagens em uma busca bem-sucedida:
Se a tabela de espalhamento estiver cheia até a metade, o número esperado de sondagens em uma busca bem-sucedida será menor que 1,387. Se a tabela de espalhamento estiver 90% cheia, o número esperado de sondagens será menor que 2,559.

Exercícios

11.4-1 Considere a inserção das chaves 10, 22, 31, 4, 15, 28, 17, 88, 59 em uma tabela de espalhamento de comprimento $m = 11$ usando endereçamento aberto com a função hash auxiliar $h'(k) = k$. Ilustre o resultado da inserção dessas chaves utilizando sondagem linear, utilizando sondagem quadrática com $c_1 = 1$ e $c_2 = 3$, e utilizando hash duplo com $h_1(k) = k$ e $h_2(k) = 1 + (k \bmod (m - 1))$.

11.4-2 Escreva pseudocódigo para HASH-DELETE como descrito no texto e modifique HASH-INSERT para manipular o valor especial DELETED.

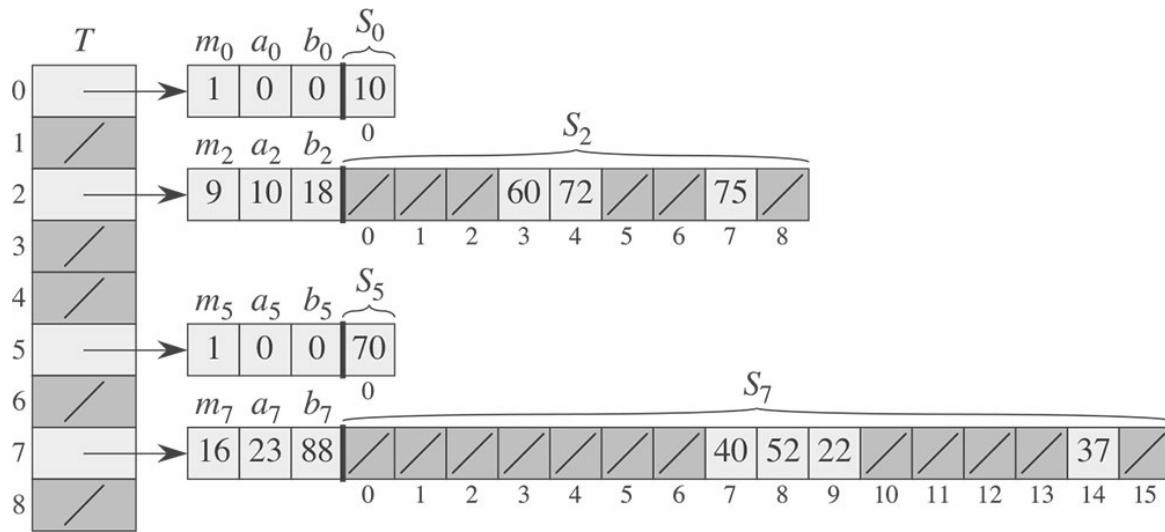
11.4-3 Considere uma tabela de espalhamento de endereços abertos com hashing uniforme. Dê limites superiores para o número esperado de sondagens em uma busca mal sucedida e para o número de sondagens em uma busca bem sucedida quando o fator de carga for 3/4 e quando for 7/8.

11.4-4 ★ Suponha que utilizamos hash duplo para resolver colisões; isto é, usamos a função hash $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Mostre que, se m e $h_2(k)$ têm máximo divisor comum $d \geq 1$ para alguma chave k, então uma busca mal sucedida para a chave k examina $(1/d)$-ésimo da tabela de espalhamento antes de retornar à posição $h_1(k)$. Assim, quando $d = 1$, de modo que m e $h_2(k)$ são primos entre si, a busca pode examinar a tabela de espalhamento inteira. (*Sugestão:* Consulte o Capítulo 31.)

11.4-5 ★ Considere uma tabela de espalhamento de endereço aberto com um fator de carga α. Encontre o valor não zero α para o qual o número esperado de sondagens em uma busca mal sucedida é igual a duas vezes o número esperado de sondagens em uma busca bem sucedida. Use os limites superiores dados pelos Teoremas 11.6 e 11.8 para esses números esperados de sondagens.

11.5 ★ HASHING PERFEITO

Embora, muitas vezes, seja uma boa escolha por seu excelente desempenho no caso médio, o hashing também pode proporcionar excelente desempenho *no pior caso*, quando o conjunto de chaves é ***estático***: assim que as chaves são armazenadas na tabela, o conjunto de chaves nunca muda. Algumas aplicações têm naturalmente conjuntos de chaves estáticos: considere o conjunto de palavras reservadas em uma linguagem de programação ou o conjunto de nomes de arquivos em um CD-ROM. Damos a uma técnica de hashing o nome de ***hashing perfeito*** se forem exigidos $O(1)$ acessos à memória para executar uma busca no pior caso.

Para criar um esquema de hashing perfeito, usamos dois níveis de aplicação do hash, com hashing universal em cada nível. A Figura 11.6 ilustra essa abordagem.

O primeiro nível é essencialmente o mesmo do hashing com encadeamento: as n chaves são espalhadas por m posições utilizando uma função hash h cuidadosamente selecionada de uma família de funções hash universal.

Porém, em vez de fazer uma lista ligada das chaves espalhadas para a posição j, usamos uma pequena ***tabela de hashing secundário*** S_j com uma função hash associada h_j. Escolhendo as funções hash h_j cuidadosamente, podemos garantir que não haverá colisões no nível secundário.

Contudo, para garantir que não haverá nenhuma colisão no nível secundário, é preciso que o tamanho m_j da tabela de espalhamento S_j seja o quadrado do número n_j de chaves que se espalham para a posição j. Embora pareça provável que a dependência quadrática de m_j em relação a n_j torne excessivo o requisito global de armazenamento, mostraremos que, escolhendo bem a função hash de primeiro nível, podemos limitar a quantidade total de espaço usado esperado a $O(n)$.

Usamos funções hash escolhidas das classes universais de funções hash da Seção 1.3.3. A função hash de primeiro nível vem da classe $_{pm}$ onde, como na Seção 1.3.3, p é um número primo maior que qualquer valor de chave. Essas chaves espalhadas para a posição j são espalhadas mais uma vez para uma tabela de espalhamento secundário S_j de tamanho m_j com a utilização de uma função hash h_j escolhida na classe $_{p,\ mj^1}$.

Figura 11.6 Utilização de hash perfeito para armazenar o conjunto $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. A função hash externa é $h(k) = ((ak + b) \bmod p) \bmod m$, onde a = 3, $b = 42$, $p = 101$, e $m = 9$. Por exemplo, $h(75) = 2$, e portanto os hashes da chave 75 vão para o espaço vazio 2 da tabela T. Uma tabela de hash secundária S_j armazena todas as chaves cujos hashes vão para o espaço vazio j. O tamanho da tabela de hash S é mj = n_2 , e a função hash associada é $h_j(k) = ((a_{jk} + b_j) \bmod p) \bmod mj$. Visto que $h_2(75) = 7$, a chave 75 é armazenada no espaço vazio 7 da tabela de hash secundária S_2 . Não ocorre nenhuma colisão em qualquer das tabelas de hash secundárias e, portanto, a busca leva tempo constante no pior caso.

Prosseguiremos em duas etapas. Primeiro, determinaremos como assegurar que as tabelas secundárias não tenham nenhuma colisão. Em segundo lugar, mostraremos que a quantidade global esperada de memória utilizada para a tabela de espalhamento primário e para todas as tabelas de espalhamento secundário é $O(n)$.

Teorema 11.9

Suponha que armazenamos n chaves em uma tabela de espalhamento de tamanho $m = n_2$ usando uma função hash h escolhida aleatoriamente de uma classe universal de funções hash. Então, a probabilidade de haver quaisquer colisões é menor que 1/2.

Prova Há pares de chaves que podem colidir; cada par colide com probabilidade $1/m$ se h é escolhida aleatoriamente de uma família universal de funções hash. Seja X uma variável aleatória que conta o número de colisões. Quando $m = n_2$, o número esperado de colisões é

$$
\begin{aligned}
E[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\
&= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\
&< 1/2.
\end{aligned}
$$

(Essa análise é semelhante à análise do paradoxo do aniversário na Seção 5.4.1.) A aplicação da desigualdade de Markov (C.30), $\Pr\{X \geq t\} \leq E[X]/t$, com $t = 1$ conclui a prova.

Na situação descrita no Teorema 11.9, onde $m = n_2$, decorre que uma função hash h escolhida aleatoriamente de tem maior probabilidade de não ter *nenhuma* colisão. Dado o conjunto K de n chaves às quais o hash deve ser aplicado (lembre-se de que K é estático), é fácil encontrar uma função hash h livre de colisões com algumas tentativas aleatórias.

Contudo, quando n é grande, uma tabela de espalhamento de tamanho $m = n_2$ é excessiva. Assim, adotamos a abordagem de hash de dois níveis e usamos a abordagem do Teorema 11.9 apenas para o hash das entradas dentro de cada posição. Usamos uma função hash h exterior, ou de primeiro nível, para espalhar as chaves para $m = n$ posições. Então, se n_j chaves são espalhadas para a posição j, usamos uma tabela de espalhamento secundário S_j de tamanho $m_j = n^2 j$ para busca de tempo constante livre de colisões.

Agora, trataremos da questão de assegurar que a memória global usada seja $O(n)$. Como o tamanho m_j da j-ésima tabela de espalhamento secundário cresce quadraticamente com o número n_j de chaves armazenadas, existe o risco de a quantidade global de armazenamento ser excessiva.

Se o tamanho da tabela de primeiro nível é $m = n$, então a quantidade de memória usada é $O(n)$ para a tabela de espalhamento primário, para o armazenamento dos tamanhos m_j das tabelas de espalhamento secundário e para o armazenamento dos parâmetros a_j e b_j que definem as funções hash secundário h_j extraídas da classe $_{p,m}$ da Seção 11.3.3 (exceto quando $n_j = 1$ e usamos $a = b = 0$). Os teorema e corolário seguintes fornecem um limite para os tamanhos combinados esperados de todas as tabelas de espalhamento secundário. Um segundo corolário limita a probabilidade de o tamanho combinado de todas as tabelas de espalhamento secundário ser superlinear. (Na verdade, equivale ou excede $4n$).

Teorema 11.10

Suponha que armazenamos n chaves em uma tabela de espalhamento de tamanho $m = n$ usando uma função hash h escolhida aleatoriamente de uma classe universal de funções hash. Então, temos

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n,$$

onde n_j é o número de chaves que efetuam hash para a posição j.

Prova Começamos com a identidade a seguir, válida para qualquer inteiro não negativo a:

$$a^2 = a + 2\binom{a}{2}. \quad (11.6)$$

Temos

$$
\begin{aligned}
E\left[\sum_{j=0}^{m-1} n_j^2\right] & \\
&= E\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(pela equação (11.6))} \\
&= E\left[\sum_{j=0}^{m-1} n_j\right] + 2E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(por lineraridade da esperança)} \\
&= E[n] + 2\,E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(pela equação (11.1))} \\
&= n + E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(já que } n \text{ não é uma variável aleatória)}
\end{aligned}
$$

Para avaliar o somatório $\sum_{j=0}^{m-1}\binom{n_j}{2}$, observamos que ele é simplesmente o número total de pares de chaves na tabela de espalhamento que colidem. Pelas propriedades de hash universal, o valor esperado desse somatório é, no máximo,

$$
\begin{aligned}
\binom{n}{2}\frac{1}{m} &= \frac{n(n-1)}{2m} \\
&= \frac{n-1}{2},
\end{aligned}
$$

já que $m = n$. Assim,

$$
\begin{aligned}
E\left[\sum_{j=0}^{m-1} n_j^2\right] &\leq n + 2\frac{n-1}{2} \\
&= 2n-1 \\
&< 2n.
\end{aligned}
$$

Corolário 11.11

Suponha que armazenamos n chaves em uma tabela de espalhamento de tamanho $m = n$ usando uma função hash h escolhida aleatoriamente de uma classe universal de funções hash e definimos o tamanho de cada tabela de espalhamento secundário como $m_j = n^2{}_j$ para $j = 0, 1, ..., m - 1$. Então, a quantidade esperada de armazenamento exigido para todas as tabelas de espalhamento secundário em um esquema de hash perfeito é menor que $2n$.

Prova Visto que $m_j = n^2{}_j$ para $j = 0, 1, ..., m - 1$, o Teorema 11.10 nos dá

$$
\begin{aligned}
E\left[\sum_{j=0}^{m-1} m_j\right] &= E\left[\sum_{j=0}^{m-1} n_j^2\right] \\
&< 2n. && (11.7)
\end{aligned}
$$

o que conclui a prova.

Corolário 11.12

Suponha que armazenamos n chaves em uma tabela de espalhamento de tamanho $m = n$, usando uma função hash h escolhida aleatoriamente de uma classe universal de funções hash e definimos o tamanho de cada tabela de espalhamento secundário como $m_j = n_j^2$ para $j = 0, 1, ..., m - 1$. Então, a probabilidade de o armazenamento total usado para tabelas de espalhamento secundário ser igual ou exceder $4n$ é menor que 1/2.

Prova Aplicamos mais uma vez a desigualdade de Markov (C.30), $\Pr\{X \geq t\} \leq E[X]/t$, desta vez à desigualdade (11.7), com $X = \sum_{j=0}^{m-1}$ e $t = 4n$:

$$\begin{aligned} \Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} &\leq \frac{E\left[\sum_{j=0}^{m-1} m_j\right]}{4n} \\ &< \frac{2n}{4n} \\ &= 1/2. \end{aligned}$$

Pelo Corolário 11.12, vemos que, se testarmos algumas funções hash escolhidas aleatoriamente de uma família universal, encontraremos rapidamente uma função que utiliza uma quantidade razoável de espaço de armazenamento.

Exercícios

11.5-1 ★ Suponha que inserimos n chaves em uma tabela de espalhamento de tamanho m usando o endereçamento aberto e hash uniforme. Seja $p(n, m)$ a probabilidade de não ocorrer nenhuma colisão. Mostre que $p(n, m) \leq e^{-n(n-1)/2m}$. (*Sugestão*: Consulte a equação (3.12).) Demonstre que, quando n excede $\sqrt{m}$, a probabilidade de evitar colisões cai rapidamente a zero.

Problemas

11-1 ***Limite para sondagem mais longa em hashing***

Suponha que usamos uma tabela de espalhamento de endereçamento aberto de tamanho m para armazenar $n \leq m/2$. itens.

a. Considerando hashing uniforme, mostre que, para $i = 1, 2, ..., n$, a probabilidade de a i-ésima inserção exigir estritamente mais de k sondagens é, no máximo, 2^{-k}.

b. Mostre que, para $i = 1, 2, ..., n$, a probabilidade de a i-ésima inserção exigir mais de $2 \lg n$ sondagens é, no máximo, $O(1/n^2)$.

Seja X_i a variável aleatória que denota o número de sondagens exigidas pela i-ésima inserção. Você mostrou na parte (b) que $\Pr\{X_i > 2 \lg n\} \leq O(1/n^2)$. Seja $X = \max_{1 \leq i \leq n} X_i$ a variável aleatória que denota o número máximo de sondagens exigidas por quaisquer das n inserções.

c. Mostre que $\Pr\{X > 2 \lg n\} \leq O(1/n)$.

d. Mostre que o comprimento esperado EX da sequência de sondagens mais longa é $O(\lg n)$.

11-2 Limite do tamanho por posição no caso de encadeamento

Suponha que temos uma tabela de espalhamento com n posições, com colisões resolvidas por encadeamento e que n chaves sejam inseridas na tabela. Cada chave tem igual probabilidade de ter hash para cada posição. Seja M o número máximo de chaves em qualquer posição após todas as chaves terem sido inseridas. Sua missão é provar um limite superior $O(\lg n/\lg \lg n)$ para E[M], o valor esperado de M.

a. Mostre que a probabilidade Q_k de ocorrer hash de exatamente k chaves para uma determinada posição é dada por

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

b. Seja P_k a probabilidade de $M = k$, isto é, a probabilidade de a posição que contém o número máximo de chaves conter k chaves. Mostre que $P_k \leq nQ_k$.

c. Use a aproximação de Stirling, equação (3.18), para mostrar que $Q_k < e^k / k^k$.

d. Mostre que existe uma constante $c > 1$ tal que $Q_{k0} < 1/n_3$ para $k_0 = c \lg n/\lg \lg n$. Conclua que $P_k < 1/n^2$ para $k \geq k_0 = c \lg n/\lg \lg n$.

e. Mostre que

$$E[M] \leq \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \leq \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n}.$$

Conclua que $E[M] = O(\lg n/\lg \lg n)$.

11-3 Sondagem quadrática

Suponha que recebemos uma chave k para buscar numa tabela de espalhamento com posições 0, 1, ..., $m - 1$, e suponha que temos uma função hash h que mapeia o espaço de chaves para o conjunto $\{0, 1, ..., m - 1\}$. O esquema de busca é o seguinte:

1. Calcule o valor $j = h(k)$ e faça $i = 0$.

2. Sonde a posição j em busca da chave desejada k. Se a encontrar ou se essa posição estiver vazia, encerre a busca.

3. Faça $j = i + 1$. Se agora i for igual a m, a tabela está cheia; portanto, encerre a busca. Caso contrário, defina $j = (i + j) \bmod m$ e retorne à etapa 2.

Suponha que m é uma potência de 2.

a. Mostre que esse esquema é uma instância do esquema geral de "sondagem quadrática", exibindo as constantes c_1 e c_2 adequadas para a equação (11.5).

b. Prove que esse algoritmo examina cada posição da tabela no pior caso.

11-4 Hashing e autenticação

Seja uma classe de funções hash na qual cada função hash $h \in$ mapeia o universo U de chaves para $\{0, 1, ..., m-1\}$. Dizemos que H é ***k-universal*** se, para toda sequência fixa de k chaves distintas $\langle x_{(1)}, x_{(2)}, ..., x_{(k)}\rangle$ e para qualquer h escolhido aleatoriamente de H, a sequência $\langle h(x_{(1)}), h(x_{(2)}), ..., h(x_{(k)})\rangle$ tem igual probabilidade de ser qualquer uma das m_k sequências de comprimento k com elementos extraídos de $\{0, 1, ..., m-1\}$.

a. Mostre que, se a família H de funções hash é 2-universal, então ela é universal.

b. Suponha que o universo U seja o conjunto de ênuplas de valores extraídos de $_p$ $\{0,1,... p-1\}$, onde p é primo. Considere um elemento $x = \langle x_0 x_1,...,x_{n-1}\rangle \in U$. Para qualquer ênupla $a = \langle a_0 a_1,...,a_{n-1}\rangle \in U$, defina a função hash h_a por

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j\right) \bmod p$$

Seja $= \{h_a\}$. Mostre que é universal, mas não 2-universal. (Sugestão: Encontre uma chave para a qual todas as funções hash em produzem o mesmo valor.)

c. Suponha que modificamos ligeiramente em relação à parte (b): para qualquer $a \in U$ e para qualquer $b \in {}_p$, defina

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b\right) \bmod p$$

e $' = \{h'_{ab}\}$. Demonstre que $'$ é 2-universal. (*Sugestão:* Considere ênuplas fixas $x \in U$ e $y \in U$, com $x_i \neq y_i$ para alguns i. O que acontece com $h'_{ab}(x)$ e $h'_{ab}(y)$ à medida que a_i e b percorrem $_p$?)

d. Suponha que Alice e Bob concordam secretamente com uma função hash h de uma família 2-universal de funções hash. Cada $h \in$ mapeia de um universo de chaves U para $_p$, onde p é primo. Mais tarde, Alice envia uma mensagem m a Bob pela Internet, na qual $m \in U$. Ela autentica essa mensagem para Bob também enviando uma marca de autenticação $t = h(m)$, e Bob verifica se o par (m, t) que ele recebe satisfaz $t = h(m)$. Suponha que um adversário intercepte (m, t) em trânsito e tente enganar Bob substituindo o par (m, t) que recebeu por um par diferente (m, t). Mostre que a probabilidade de o adversário conseguir enganar Bob e fazê-lo aceitar (m, t) é, no máximo, $1/p$, independentemente da capacidade de computação que o adversário tenha e até mesmo de o adversário conhecer a família de funções hash usada.

NOTAS DO CAPÍTULO

Knuth [211] e Gonnet [145] são excelentes referências para a análise de algoritmos de hashing. Knuth credita a H. P. Luhn (1953) a criação de tabelas de espalhamento, juntamente com o método de encadeamento para resolver colisões. Aproximadamente na mesma época, G. M. Amdahl apresentou a ideia do endereçamento aberto.

Carter e Wegman introduziram a noção de classes universais de funções hash em 1979 [58]. Fredman, Komlós e Szemerédi [112] desenvolveram o esquema de hashing perfeito para conjuntos estáticos apresentado na Seção 11.5.

Uma extensão de seu método para conjuntos dinâmicos, tratamento de inserções e eliminações em tempo esperado amortizado $O(1)$, foi apresentada por Dietzfelbinger *et al*. [87].

---

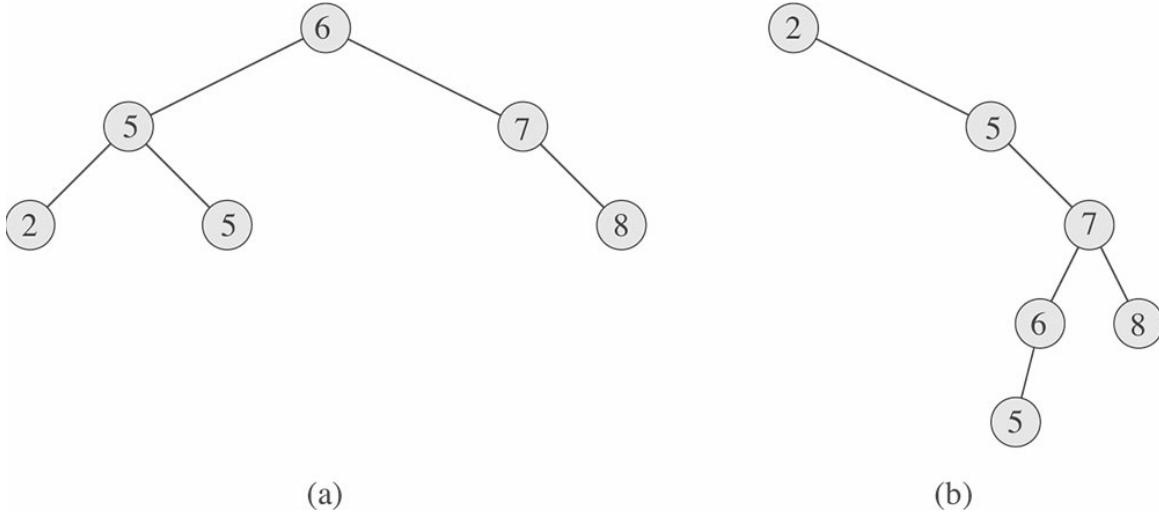
[1] Quando $n = m = 1$, não precisamos realmente de uma função hash para a posição j; quando escolhemos uma função hash $h(k) = ((ak + b) \bmod p) \bmod m_j$ para tal posição, usamos simplesmente $a = b = 0$.

12 ÁRVORES DE BUSCA BINÁRIA

A estrutura árvore de busca suporta muitas operações de conjuntos dinâmicos, incluindo SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT e DELETE. Assim, uma árvore de busca pode ser usada como um dicionário e também como uma fila de prioridades.

As operações básicas em uma árvore de busca binária demoram um tempo proporcional à altura da árvore. No caso de uma árvore binária completa com n nós, tais operações são executadas no tempo $Q(\lg n)$ do pior caso. Porém, se a árvore é uma cadeia linear de n nós, as mesmas operações demoram o tempo $Q(n)$ do pior caso. Veremos na Seção 12.4 que a altura esperada de uma árvore de busca binária construída aleatoriamente é $O(\lg n)$, de modo que as operações básicas de conjuntos dinâmicos em tal árvore demoram o tempo $Q(\lg n)$ em média.

Na prática, nem sempre podemos garantir que as árvores de busca binária sejam construídas aleatoriamente, mas podemos projetar variações de árvores de busca binária com bom desempenho do pior caso garantido em operações básicas. O Capítulo 13 apresenta uma dessas variações, as árvores vermelhopreto, que têm altura $O(\lg n)$. O Capítulo 18 introduz as árvores B (Btrees), que são particularmente boas para manter bancos de dados em armazenamento secundário (disco).

Depois da apresentação das propriedades básicas de árvores de busca binária, as próximas seções mostram como percorrer uma árvore de busca binária para imprimir seus valores em sequência ordenada, como procurar um valor em uma árvore de busca binária, como encontrar o elemento mínimo ou máximo, como encontrar o predecessor ou o sucessor de um elemento e como inserir ou eliminar elementos em uma árvore de busca binária. As propriedades matemáticas básicas das árvores são apresentadas no Apêndice B.

12.1 O QUE É UMA ÁRVORE DE BUSCA BINÁRIA?

Uma árvore de busca binária é organizada, como o nome sugere, em uma árvore binária, como mostra a Figura 12.1. Podemos representar tal árvore por uma estrutura de dados ligada, na qual cada nó é um objeto. Além de uma *chave* e de dados satélites, cada nó contém atributos *esquerda*, *direita* e *p*, que apontam para os nós correspondentes ao seu filho à esquerda, ao seu filho à direita e ao seu pai, respectivamente. Se um filho ou o pai estiver ausente, o atributo adequado contém o valor NIL. O nó raiz é o único nó na árvore cujo pai é NIL.

As chaves em uma árvore de busca binária são sempre armazenadas de modo a satisfazer a propriedade de árvore de busca binária:

Seja x um nó em uma árvore de busca binária. Se y é um nó na subárvore esquerda de x, então $y.chave \leq x.chave$. Se y é um nó na subárvore direita de x, então $x.chave \geq y.chave$.

Assim, na Figura 12.1(a), a chave da raiz é 6, as chaves 2, 5 e 5 em sua subárvore esquerda não são maiores que 6, e as chaves 7 e 8 em sua subárvore direita não são menores que 6. A mesma propriedade é válida para todo nó na árvore. Por exemplo, a chave 5 no filho à esquerda da raiz não é menor que a chave 2 na subárvore esquerda daquele nó e não é maior que a chave 5 na subárvore direita.

Figura 12.1 Árvores de busca binária. Para qualquer nó x, as chaves na subárvore esquerda de x são no máximo $x.chave$, e as chaves na subárvore direita de x são no mínimo $x.chave$. Árvores de busca binária diferentes podem representar o mesmo conjunto de valores. O tempo de execução do pior caso para a maioria das operações em árvores de busca é proporcional à altura da árvore. (**a**) Uma árvore de busca binária com seis nós e altura 2. (**b**) Uma árvore de busca binária menos eficiente, com altura 4, que contém as mesmas chaves.

A propriedade de árvore de busca binária nos permite imprimir todas as chaves em uma árvore de busca binária em sequência ordenada por meio de um simples algoritmo recursivo, denominado ***percurso de árvore em in-ordem***. Esse algoritmo tem tal nome porque imprime a chave da raiz de uma subárvore entre a impressão dos valores em sua subárvore esquerda e a impressão dos valores em sua subárvore direita. (De modo semelhante, um ***percurso de árvore em pré-ordem*** imprime a raiz antes dos valores das subárvores, e um ***percurso de árvore em pós-ordem*** imprime a raiz depois dos valores em suas subárvores.) Para usar o procedimento a seguir com o objetivo de imprimir todos os elementos em uma árvore de busca binária T, chamamos INORDER-TREE-WALK($T.raiz$).

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.esquerda)
3      print x.chave
4      INORDER-TREE-WALK(x.direita)
```

Como exemplo, o percurso de árvore em ordem imprime as chaves em cada uma das duas árvores de busca binária da Figura 12.1 na ordem 2, 3, 5, 6, 7, 8. A correção do algoritmo decorre por indução diretamente da propriedade de árvore de busca binária.

Percorrer uma árvore de busca binária de n nós demora o tempo Q(n) já que, após a chamada inicial, o procedimento chama a si mesmo recursivamente exatas duas vezes para cada nó na árvore — uma vez para seu filho à esquerda e uma vez para seu filho à direita. O teorema a seguir, apresenta uma prova formal de que o tempo para executar um percurso de árvore em inordem é linear.

Teorema 12.1

Se x é a raiz de uma subárvore de n nós, então a chamada INORDER-TREE-WALK(x) demora o tempo Q(n).

Prova Seja $T(n)$ o tempo tomado por INORDER-TREE-WALK quando chamado na raiz de uma subárvore de n nós. Visto que INORDER-TREE-WALK visita todos os n nós da subárvore, temos $T(n) = (n)$. Resta mostrar que $T(n) = O(n)$.

Uma vez que INORDER-TREE-WALK demora um tempo pequeno e constante em uma subárvore vazia (para o teste $x \neq$ NIL), temos $T(0) = c$ para alguma constante $c > 0$.

Para $n > 0$, suponha que INORDER-TREE-WALK seja chamado em um nó x cuja subárvore esquerda tem k nós e cuja subárvore direita tem $n - k - 1$ nós. O tempo para executar INORDER-TREE-WALK(x) é limitado por $T(n) = T(k) + T(n - k - 1) + d$ para alguma constante d > 0, que reflete um limite superior para o tempo de execução do corpo de INORDER-TREE-WALK(x), excluindo o tempo gasto em chamadas recursivas.

Usamos o método de substituição para mostrar que $T(n)$ = Q(n), provando que $T(n) \leq (c + d)n + c$. Para $n = 0$, temos $(c + d) \cdot 0 + c = c = T(0)$. Para $n > 0$, temos

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c \ , \end{aligned}$$

o que conclui a prova.

Exercícios

12.1-1 Trace árvores de busca binária de alturas 2, 3, 4, 5 e 6 para o conjunto de chaves $\{1, 4, 5, 10, 16, 17, 21\}$.

12.1-2 Qual é a diferença entre a propriedade de árvore de busca binária e a propriedade de heap de mínimo (veja p. 153)? A propriedade de heap de mínimo pode ser usada para imprimir as chaves de uma árvore de n nós em sequência ordenada no tempo $O(n)$? Justifique sua resposta.

12.1-3 Dê um algoritmo não recursivo que execute um percurso de árvore em ordem. (*Sugestão:* Uma solução fácil usa uma pilha como uma estrutura de dados auxiliar. Uma solução mais complicada, porém elegante, não emprega nenhuma pilha, mas considera que é possível testar a igualdade entre dois ponteiros.)

12.1-4 Dê algoritmos recursivos que executem percursos de árvores em préordem e pós-ordem no tempo Q(n) em uma árvore de n nós.

12.1-5 Mostre que, considerando que a ordenação de n elementos demora o tempo (n lg n) no pior caso do modelo de comparação, qualquer algoritmo baseado em comparação para construir uma árvore de busca binária com base em uma lista arbitrária de n elementos demora o tempo (n lg n) no pior caso.

12.2 CONSULTAS EM UMA ÁRVORE DE BUSCA BINÁRIA

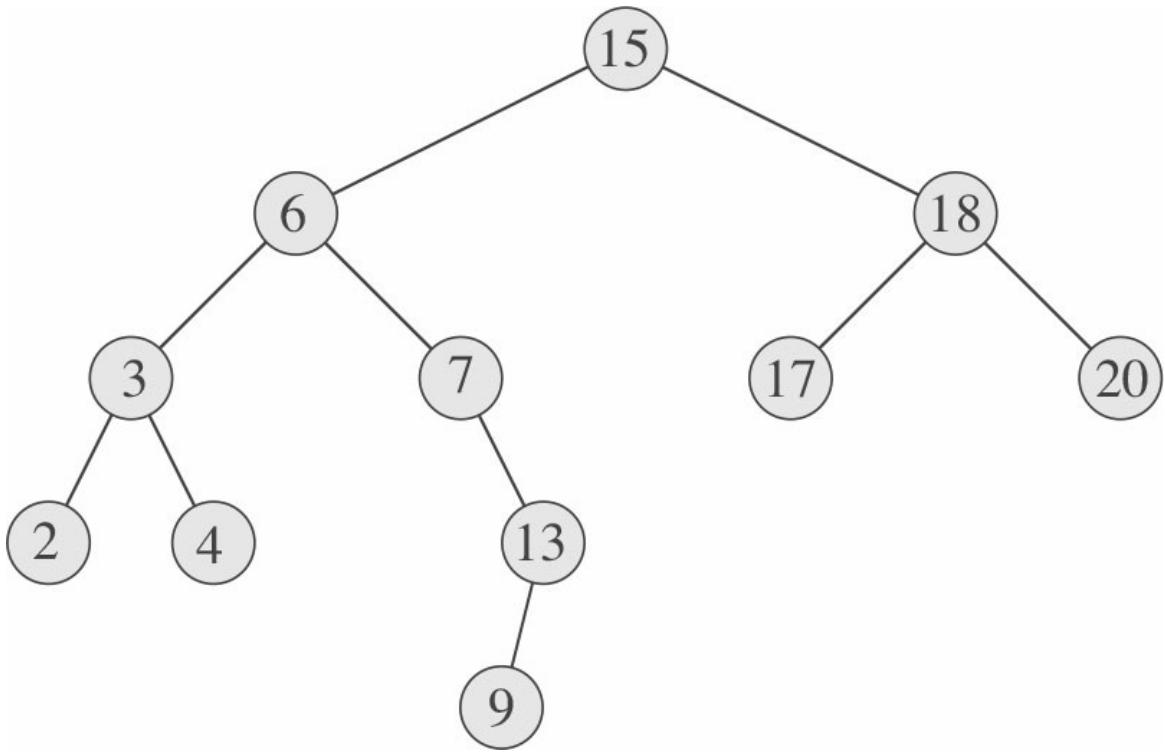
Frequentemente precisamos procurar uma chave armazenada em uma árvore de busca binária. Além da operação SEARCH, árvores de busca binária podem suportar as consultas MINIMUM, INOR- DER-MAXIMUM, SUCCESSOR e PREDECESSOR. Nesta seção, examinaremos essas operações e mostraremos como suportar cada uma delas no tempo $O(h)$ em qualquer árvore de busca binária de altura h.

Buscas

Usamos o procedimento a seguir para procurar um nó com determinada chave em uma árvore de busca binária. Dado um ponteiro para a raiz da árvore e uma chave k, TREE-SEARCH retorna um ponteiro para um nó com chave k, se existir algum; caso contrário, ele retorna NIL.

```
TREE-SEARCH(x, k)
1  if x == NIL ou k == x.chave
2      return x
3  if k < x.chave
4      return TREE-SEARCH(x.esquerda, k)
5  else return TREE-SEARCH(x.direita, k)
```

O procedimento começa sua busca na raiz e traça um caminho simples descendo a árvore, como mostra a Figura 12.2. Para cada nó x que encontra, ele compara a chave k com a $x.chave$. Se as duas chaves são iguais, a busca termina. Se k é menor que $x.chave$, a busca continua na subárvore esquerda de x, já que a propriedade de árvore de busca binária implica que k não poderia estar armazenada na subárvore direita. Simetricamente, se k é maior que $x.chave$, a busca continua na subárvore direita. Os nós encontrados durante a recursão formam um caminho simples descendente partindo da raiz da árvore e, portanto, o tempo de execução de TREE-SEARCH é $O(h)$, onde h é a altura da árvore.

O mesmo procedimento pode ser reescrito de modo iterativo, "desdobrando" a recursão dentro de um laço **while**. Na maioria dos computadores, a versão iterativa é mais eficiente.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL e k ≠ x.chave
2     if k < x.chave
3            x = x.esquerda
4     else x = x.direita
5  return x
```

Mínimo e máximo

Sempre podemos encontrar um elemento em uma árvore de busca binária cuja chave é um mínimo seguindo ponteiros de filhos da *esquerda* desde a raiz até encontrarmos um valor NIL, como mostra a Figura 12.2. O procedimento a seguir, retorna um ponteiro para o elemento mínimo na subárvore enraizada em um nó x dado.

Figura 12.2 Consultas em uma árvore de busca binária. Para procurar a chave 13 na árvore, seguimos o caminho 15 → 6 → 7 → 13 partindo da raiz. A chave mínima na árvore é 2, que é encontrada seguindo os ponteiros da *esquerda* partindo da raiz. A chave máxima 20 é encontrada seguindo os ponteiros da *direita* partindo da raiz. O sucessor do nó com chave 15 é o nó com chave 17, já que ele é a chave mínima na subárvore direita de 15. O nó com chave 13 não tem nenhuma subárvore direita e, assim, seu sucessor é seu ancestral mais baixo cujo filho à esquerda também é um ancestral. Nesse caso, o nó com chave 15 é seu sucessor.

```
TREE-MINIMUM(x)
1   while x.esquerda ≠ NIL
2       x = x.esquerda
3   return x
```

A propriedade de árvore de busca binária garante que TREE-MINIMUM é correto. Se um nó x não tem nenhuma subárvore esquerda, então, visto que toda chave na subárvore direita de x é no mínimo tão grande quanto $x.chave$, a chave mínima na subárvore enraizada em x é $x.chave$. Se o nó x tem uma subárvore esquerda, então, visto que nenhuma chave na subárvore direita é menor que $x.chave$ e toda chave na subárvore esquerda não é maior que $x.chave$, a chave mínima na subárvore enraizada em x pode ser encontrada na subárvore enraizada em $x.esquerda$.

O pseudocódigo para TREE-MAXIMUM é simétrico.

```
TREE-MAXIMUM(x)
1   while x.direita ≠ NIL
2       x = x.direita
3   return x
```

Ambos os procedimentos são executados no tempo $O(h)$ em uma árvore de altura h já que, como em TREE-SEARCH, a sequência de nós encontrados forma um caminho simples descendente partindo da raiz.

Sucessor e predecessor

Dado um nó em uma árvore de busca binária, às vezes, precisamos encontrar seu sucessor na sequência ordenada determinada por um percurso de árvore em ordem. Se todas as chaves são distintas, o sucessor de um nó x é o nó com a menor chave maior que $chave[x]$. A estrutura de uma árvore de busca binária nos permite determinar o sucessor de um nó sem sequer comparar chaves. O procedimento a seguir, retorna o sucessor de um nó x em uma árvore de busca binária se ele existir, e NIL se x tem a maior chave na árvore.

```
TREE-SUCCESSOR(x)

1  if x.direita ≠ NIL
2      return TREE-MINIMUM(x.direita)
3  y = p[x]
4  while y ≠ NIL e x = y.direita
5      x = y
6      y = y.p
7  return y
```

Subdividimos o código para TREE-SUCCESSOR em dois casos. Se a subárvore direita do nó x for não vazia, então o sucessor de x é exatamente o nó da extrema esquerda na subárvore direita de x, que encontramos na linha 2 chamando TREE-MINIMUM($x.direita$). Por exemplo, o sucessor do nó com chave 15 na Figura 12.2 é o nó com chave 17.

Por outro lado, como o Exercício 12.26 pede que você mostre, se a subárvore direita do nó x é vazia e x tem um sucessor y, então y é o ancestral mais baixo de x cujo filho à esquerda é também um ancestral de x. Na Figura 12.2, o sucessor do nó com chave 13 é o nó com chave 15. Para encontrar y, simplesmente subimos a árvore desde x até encontrarmos um nó que seja o filho à esquerda de seu pai; isso é conseguido por meio das linhas 3 a 7 de TREE-SUCCESSOR.

O tempo de execução de TREE-SUCCESSOR em uma árvore de altura h é $O(h)$, já que seguimos um caminho simples para cima na árvore ou, então, um caminho simples para baixo na árvore. O procedimento TREE-PREDECESSOR, que é simétrico de TREE-SUCCESSOR, também é executado no tempo $O(h)$.

Ainda que as chaves não sejam distintas, definimos o sucessor e o predecessor de qualquer nó x como o nó retornado por chamadas feitas a TREE-SUCCESSOR(x) e TREE-PREDECESSOR(x), respectivamente.

Resumindo, demonstramos o teorema a seguir.

Teorema 12.2

Podemos implementar as operações de conjuntos dinâmicos SEARCH, MINIMUM, MAXIMUM, TREE-SUCCESSOR e PREDECESSOR de modo que cada uma seja executada no tempo $O(h)$ em uma árvore de busca binária de altura h.

Exercícios

12.2-1 Suponha que temos números entre 1 e 1.000 em uma árvore de busca binária e queremos procurar o número 363. Qual das seguintes sequências *não* poderia ser a sequência de nós examinados?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2 Escreva versões recursivas de TREE-MINIMUM e TREE-MAXIMUM

12.2-3 Escreva o procedimento TREE-PREDECESSOR.

12.2-4 O professor Bunyan pensa ter descoberto uma notável propriedade de árvores de busca binária. Suponha que a busca da chave k em uma árvore de busca binária termine em uma folha. Considere três conjuntos: A, as chaves à esquerda do caminho de busca; B, as chaves no caminho de busca; e C, as chaves à direita do caminho de busca. O professor Bunyan afirma que quaisquer três chaves $a \in A$, $b \in B$ *e* $c \in C$ devem satisfazer $a \;\; b \leq c$. Dê um contraexemplo menor possível para a afirmação do professor.

12.2-5 Mostre que, se um nó em uma árvore de busca binária tem dois filhos, então seu sucessor não tem nenhum filho à esquerda e seu predecessor não tem nenhum filho à direita.

12.2-6 Considere uma árvore de busca binária T cujas chaves são distintas. Mostre que, se a subárvore direita de um nó x em T é vazia e x tem um sucessor y, então y é o ancestral mais baixo de x cujo filho à esquerda também é um ancestral de x. (Lembrese de que todo nó é seu próprio ancestral.)

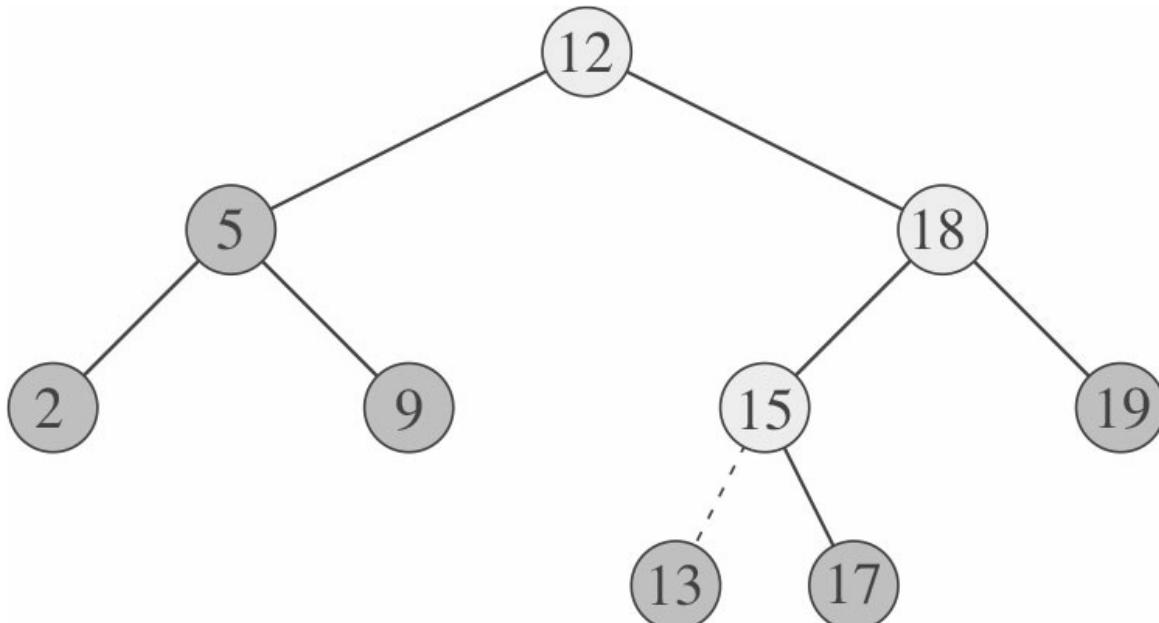
12.2-7 Um método alternativo de executar um percurso de árvore em ordem em uma árvore de busca binária de n nós encontra o elemento mínimo na árvore chamando TREE-MINIMUM e depois fazendo $n - 1$ chamadas a TREE-SUCCESSOR. Prove que esse algoritmo é executado no tempo Q(n).

12.2-8 Prove que, independentemente do nó onde iniciamos em uma árvore de busca binária de altura h, k chamadas sucessivas a TREE-SUCCESSOR demoram o tempo $O(k + h)$.

12.2-9 Seja T uma árvore de busca binária cujas chaves são distintas, seja x um nó de folha e seja y seu pai. Mostre que *y.chave* é a menor chave em T maior que $x.chave$ ou a maior chave em T menor que $x.chave$.

12.3 INSERÇÃO E ELIMINAÇÃO

As operações de inserção e eliminação provocam mudanças no conjunto dinâmico representado por uma árvore de busca binária. A estrutura de dados deve ser modificada para refletir essa mudança, mas de tal modo que a propriedade de árvore de busca binária continue válida. Como veremos, modificar a árvore para inserir um novo elemento é uma operação relativamente direta, mas manipular a eliminação é um pouco mais complicado.

Inserção

Para inserir um novo valor v em uma árvore de busca binária T, utilizamos o procedimento TREE-INSERT. O procedimento toma um nó z para o qual $z.chave = v$, $z.esquerda$ = NIL e $z.direita$ = NIL, e modifica T e alguns dos atributos de z de modo tal que insere z em uma posição adequada na árvore.

```
TREE-INSERT(T, z)
1   y = NIL
2   x = T.raiz
3   while x ≠ NIL
4      y = x
5      if z.chave < x.chave
6               x = x.esquerda
7      else x = x.direita
8   z.p = y
9   if y = NIL
10     T.raiz = z                  //a árvore T era vazia
11  else if z.chave < y.chave
12     y.esquerda = z
13  else y.direita = z
```

A Figura 12.3 mostra como TREE-INSERT funciona. Exatamente como os procedimentos TREE-SEARCH e ITERATIVETREE-SEARCH, TREE-INSERRT começa na raiz da árvore e o ponteiro x e traça um caminho simples descendente procurando um NIL para substituir pelo item de entrada z. O procedimento mantém o ***ponteiro acompanhante*** y como o pai de x. Após a inicialização, o loop **while** nas linhas 3−7 faz com que esses dois ponteiros se desloquem para baixo na árvore, indo para a esquerda ou a direita, dependendo da comparação de $z.chave$ com $x.chave$, até x tornarse NIL. Esse NIL ocupa a posição na qual desejamos colocar o item de entrada z.

Figura 12.3 Inserção de um item com chave 13 em uma árvore de busca binária. Os nós sombreados em tom mais claro indicam o caminho simples da raiz até a posição em que o item é inserido. A linha tracejada indica a ligação que é acrescentada à árvore para inserir o item.

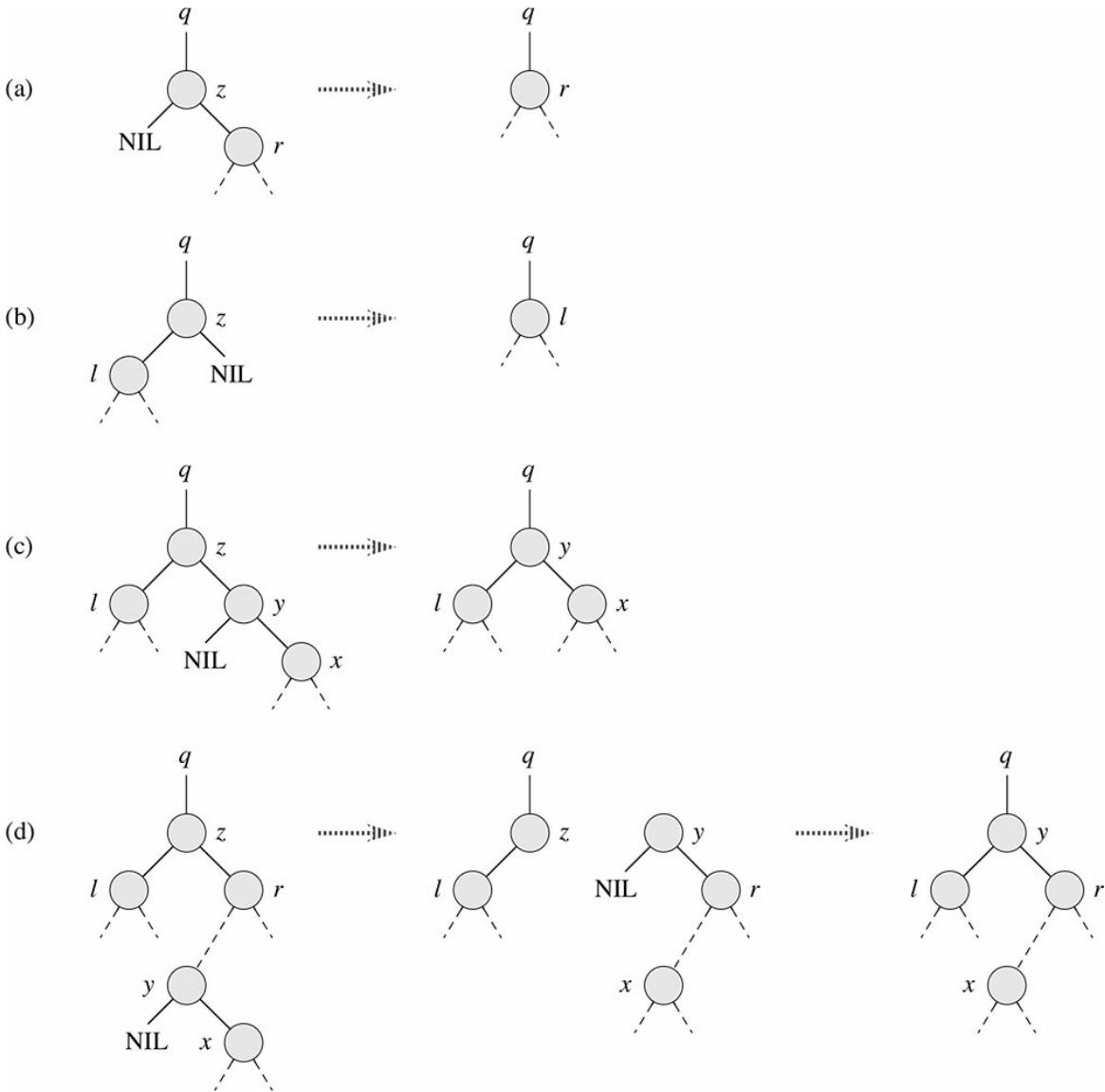
Precisamos do ponteiro acompanhante y porque, até encontrarmos o NIL ao qual z pertence, a busca desenvolveuse sempre uma etapa à frente do nó que precisa ser mudado. As linhas 8−13 definem os ponteiros que causam a inserção de z.

Do mesmo modo que as outras operações primitivas em árvores de busca, o procedimento TREE-INSERT é executado no tempo $O(h)$ em uma árvore de altura h.

Eliminação

A estratégia global para eliminar um nó z de uma árvore de busca binária T tem três casos básicos, mas, como veremos, um deles é um pouco complicado.

- Se z não tem nenhum filho, então simplesmente o removemos modificando seu pai de modo a substituir z por NIL como seu filho.
- Se o nó tem apenas um filho, então elevamos esse filho para que ocupe a posição de z na árvore modificando o pai de z de modo a substituir z pelo filho de z.
- Se z tiver dois filhos, encontramos o sucessor de z, y, que deve estar na subárvore direita de z, e obrigamos y a tomar a posição de z na árvore. O resto da subárvore direita original de z tornase a nova subárvore direita de y, e a subárvore esquerda de z tornase a nova subárvore esquerda de y. Esse é o caso complicado porque, como veremos, o fato de y ser ou não o filho à direita de z é importante.

O procedimento para eliminar um dado nó z de uma árvore de busca binária T toma como argumentos ponteiros para T e z. Esse procedimento organiza seus casos de um modo um pouco diferente dos três casos descritos, já que considera os quatro casos mostrados na Figura 12.4.

- Se z não tiver nenhum filho à esquerda (parte (a) da figura), substituímos z por seu filho à direita, que pode ser ou não NIL. Quando o filho à direita de z é NIL, esse caso trata da situação na qual z não tem nenhum filho. Quando o filho à direita de z é não NIL, esse caso manipula a situação na qual z tem somente um filho, que é o seu filho à direita.
- Se z tiver apenas um filho, que é o seu filho à esquerda (parte (b) da Figura), substituímos z por seu filho à esquerda.
- Caso contrário, z tem um filho à esquerda e também um filho à direita. Encontramos o sucessor de z, y, que está na subárvore direita de z e não tem nenhum filho à esquerda (veja o Exercício 12.25). Queremos recortar y de sua localização atual e fazêlo substituir z na árvore.
  - Se y é o filho à direita de z (parte (c)), substituímos z por y, deixando o filho à direita de y sozinho.
  - Caso contrário, y encontrase dentro da subárvore direita de z, mas não é o filho à direita de z (parte (d)). Nesse caso, primeiro substituímos y por seu próprio filho à direita e depois substituímos z por y.

Para movimentar subárvores dentro da árvore de busca binária, definimos uma subrotina TRANSPLANT, que substitui uma subárvore como um filho de seu pai por uma outra subárvore. Quando TRANSPLANT substitui a subárvore enraizada no nó u pela subárvore enraizada no nó v, o pai do nó u tornase o pai do nó v, e o pai de u acaba ficando com v como seu filho adequado.

```
TRANSPLANT (T, u, v)
1  if u.p == NIL
2      T.raiz = v
3  elseif u == u.p.esquerda
4      u.p. esquerda = v
5  else u.p.direita = v
6  if v ≠ NIL
7      v.p = u.p
```

Figura 12.4 Eliminação de um nó de uma árvore de busca binária. O nó z pode ser a raiz, um filho à esquerda do nó q ou um filho à direita de q. **(a)** O nó z não tem nenhum filho à esquerda. Substituímos z por seu filho à direita r, que pode ou não ser NIL. **(b)** O nó z tem um filho à esquerda l mas nenhum filho à direita. Substituímos z por l. **(c)** O nó z tem dois filhos; seu filho à esquerda é o nó l, seu filho à direita é seu sucessor y, e o filho à direita de y é o nó x. Substituímos z por y, atualizando o filho à esquerda de y para que se torne l, mas deixando x como o filho à direita de y. **(d)** O nó z tem dois filhos (o filho à esquerda l e o filho à direita r), e seu sucessor $y \neq$ r encontra-se dentro da subárvore enraizada em r. Substituímos y por seu próprio filho à direita x, e definimos y como pai de r. Então, tomamos y filho de q e pai de l.

As linhas 1−2 tratam do caso no qual u é a raiz de T. Caso contrário, u é um filho à esquerda ou um filho à direita de seu pai. As linhas 3−4 encarregamse de atualizar *u.p.direita* se u é um filho à esquerda, e a linha 5 atualiza *u.p.esquerda* se u é um filho à direita. Permitimos que v seja NIL, e as linhas 6−7 atualizam *v.p* se v é não NIL. Observe que TRANSPLANT não tenta atualizar update *v.esquerda e v.direita*; fazer ou não fazer isso é responsabilidade do chamador de TRANSPLANT.

Com o procedimento TRANSPLANT em mãos, apresentamos a seguir o procedimento que elimina o nó z da árvore de busca binária T:

```
TREE-DELETE (T,z)
1   if z.esquerda == NIL
2      TRANSPLANT (T, z, z.direita)
3   elseif z.direita == NIL
4      TRANSPLANT (T, z, z.esquerda)
5   else y = TREE-MINIMUM (z.direita)
6      if y.p ≠ z
7            TRANSPLANT (T, y, y.direita)
8            y.direita = z.direita
9            y.direita.p = y
10     TRANSPLANT (T, z, y)
11     y.esquerda = z.esquerda
12     y.esquerda.p = y
```

O procedimento TREE-DELETE executa os quatro casos como descrevemos a seguir. As linhas 1−2 tratam do caso no qual o nó z não tem nenhum filho à esquerda, e as linhas 3−4 tratam do caso no qual z tem um filho à esquerda mas nenhum filho à direita. As linhas 5−12 lidam com os dois casos restantes, nos quais z tem dois filhos. A linha 5 encontra o nó y, que é o sucessor de z. Como z tem uma subárvore direita não vazia, seu sucessor deve ser o nó nessa subárvore que tenha a menor chave; daí a chamada a TREE-MINIMUM ($z.direita$). Como observamos antes, y não tem nenhum filho à esquerda. Queremos recortar y de sua localização atual, e ele deve substituir z na árvore. Se y é o filho à direita de z, então as linhas 10−12 substituem z como um filho de seu pai por y e substituí o filho à esquerda de y pelo filho à esquerda de z . Se y não é o filho à direita de z , as linhas 7−9 substituem y como um filho de seu pai pelo filho à direita de y e transforma o filho à direita de z em filho à direita de y, e então as linhas 10−12 substituem z como um filho de seu pai por y e substituem o filho à esquerda de y pelo filho à esquerda de z.

Cada linha de TREE-DELETE, incluindo as chamadas a TRANSPLANT, demora tempo constante, exceto a chamada a TREE-MINIMUM na linha 5. Assim, TREE-DELETE é executado no tempo $O(h)$ em uma árvore de altura h.

Resumindo, provamos o teorema apresentado a seguir.

Teorema 12.3

Podemos implementar as operações de conjuntos dinâmicos INSERT e DELETE de modo que cada uma seja executada no tempo $O(h)$ em uma árvore de busca binária de altura h.

Exercícios

12.3-1 Dê uma versão recursiva do procedimento TREE-INSERT.

12.3-2 Suponha que construímos uma árvore de busca binária inserindo repetidamente valores distintos na árvore. Mostre que o número de nós examinados na busca de um valor na árvore é uma unidade mais o número de nós examinados quando o valor foi inicialmente inserido na árvore.

12.3-3 Podemos ordenar um dado conjunto de n números construindo primeiro uma árvore de busca binária contendo esses números (usando TREE-INSERT repetidamente para inserir os números um a um) e então

imprimindo os números por um percurso de árvore em inordem. Quais são os tempos de execução do pior caso e do melhor caso para esse algoritmo de ordenação?

12.3-4 A operação de eliminação é "comutativa" no sentido de que eliminar x e depois y de uma árvore de busca binária resulta na mesma árvore que eliminar y e depois x? Mostre por que ou dê um contraexemplo.

12.3-5 Suponha que, em vez de cada nó x manter o atributo $x.p$, apontando para o pai de x's, ele mantém $x.suc$, apontando para o sucessor de x. Dê pseudocódigo para SEARCH, INSERT e DELETE em uma árvore de busca binária T usando essa representação. Esses procedimentos devem funcionar no tempo $O(h)$, onde h é a altura da árvore T. (*Sugestão*: Seria interessante implementar uma subrotina que retorne o pai de um nó.)

12.3-6 Quando o nó z em TREE-DELETE tem dois filhos, podemos escolher o nó y como seu predecessor em vez de seu sucessor. Se fizermos isso, quais outras mudanças serão necessárias em TREE-DELETE? Há quem defenda que uma estratégia justa, que dá igual prioridade ao predecessor e ao sucessor, produz melhor desempenho empírico. Como TREE-DELETE pode ser modificado para implementar tal estratégia justa?

12.4 ★ ÁRVORES DE BUSCA BINÁRIA CONSTRUÍDAS ALEATORIAMENTE

Mostramos que cada uma das operações básicas em uma árvore de busca binária é executada no tempo $O(h)$, onde h é a altura da árvore. Contudo, a altura de uma árvore de busca binária varia à medida que itens são inseridos e eliminados. Se, por exemplo, os n itens são inseridos em ordem estritamente crescente, a árvore será uma cadeia com altura $n - 1$. Por outro lado, o Exercício B.54 mostra que $h \geq \lg n$. Como ocorre com o quicksort, podemos mostrar que o comportamento do caso médio é muito mais próximo do melhor caso que do pior caso.

Infelizmente, sabese pouco sobre a altura média de uma árvore de busca binária quando ambas, inserção e eliminação, são utilizadas para criála. Quando a árvore é criada somente por inserção, a análise se torna mais tratável. Portanto, vamos definir uma ***árvore de busca binária construída aleatoriamente*** em n chaves distintas como aquela que surge da inserção das chaves em ordem aleatória em uma árvore inicialmente vazia, onde cada uma das n! permutações das chaves de entrada é igualmente provável. (O Exercício 12.43 pede que você mostre que essa noção é diferente de considerar que toda árvore de busca binária em n chaves é igualmente provável.) Nesta seção, provaremos o teorema apresentado a seguir.

Teorema 12.4

A altura esperada de uma árvore de busca binária construída aleatoriamente em n chaves distintas é $O(\lg n)$.

Prova Começamos definindo três variáveis aleatórias que ajudam a medir a altura de uma árvore de busca binária construída aleatoriamente. Denotamos a altura de uma árvore de busca binária construída aleatoriamente em n chaves por X_n, e definimos a ***altura exponencial*** $Y_n = 2^{X_n}$. Quando construímos uma árvore de busca binária em n chaves, escolhemos uma chave como a chave da raiz e denotamos por R_n a variável aleatória que contém a ***classificação*** dessa chave dentro do conjunto de n chaves, isto é, R_n ocupa a posição que essa chave ocuparia se o conjunto de chaves fosse ordenado. O valor de R_n tem igual probabilidade de ser qualquer elemento do conjunto $\{1, 2, ..., n\}$. Se $R_n = i$, então a subárvore esquerda da raiz é uma árvore de busca binária construída aleatoriamente em $i - 1$ chaves, e a subárvore direita é uma árvore de busca binária construída aleatoriamente em $n - i$ chaves. Como a altura de uma árvore binária é uma unidade maior que a maior das alturas das duas subárvores da raiz, a altura exponencial de uma árvore binária é duas vezes a maior das alturas exponenciais das duas subárvores da raiz. Se sabemos que $R_n = i$, decorre que

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}).$$

Como casos-bases temos que $Y_1 = 1$ porque a altura exponencial de uma árvore com 1 nó é $2^0 = 1$ e, por conveniência, definimos $Y_0 = 0$.

Em seguida, definimos variáveis aleatórias indicadoras $Z_{n,1}, Z_{n,2}, ..., Z_{n,n}$, onde

$$Z_{n\,i} = \mathrm{I}\{R_n = i\}.$$

Como R_n tem igual probabilidade de ser qualquer elemento de $\{1, 2, ..., n\}$, decorre que $\Pr\{R_n = i\} = 1/n$ para $i = 1, 2, ..., n$ e, consequentemente, pelo Lema 5.1, temos

$$\mathrm{E}[Z_{n,i}] = 1/n\,, \tag{12.1}$$

para $i = 1, 2, ..., n$. Como exatamente um valor de $Z_{n,i}$ é 1 e todos os outros são 0, também temos

$$Y_n = \sum_{i=1}^{n} Z_{n,i}\ (2 \cdot \max(Y_{i-1}, Y_{n-1})).$$

Mostraremos que $\mathrm{E}[Y_n]$ é um polinômio em n, que, em última análise, implicará $\mathrm{E}[X_n] = O(\lg n)$.

Afirmamos que a variável aleatória indicadora $Z_{n,i} = \mathrm{I}\{R_n = i\}$ é independente dos valores de Y_{i-1} e Y_{n-i}. Como escolhemos $R_n = i$, a subárvore esquerda (cuja altura exponencial é Y_{i-1}) é construída aleatoriamente nas $i - 1$ chaves cujas classificações são menores que 1. Essa subárvore é exatamente igual a qualquer outra árvore de busca binária construída aleatoriamente em $i - 1$ chaves. Exceto pelo número de chaves que contém, a estrutura dessa subárvore não é afetada de modo algum pela escolha de $R_n = i$; consequentemente, as variáveis aleatórias Y_{i-1} e $Z_{n,i}$ são independentes. Do mesmo modo, a subárvore direita, cuja altura exponencial é Y_{n-i}, é construída aleatoriamente nas $n - i$ chaves cujas classificações são maiores que i. Sua estrutura é independente do valor de R_n e, assim, as variáveis aleatórias Y_{n-i} e $Z_{n,i}$ são independentes. Daí, temos

$$\begin{aligned}
E[Y_n] &= E\left[\sum_{i=1}^{n} Z_{n,i}\ (2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\
&= \sum_{i=1}^{n} E[Z_{n,i}(2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(por linearidade da esperança)} \\
&= \sum_{i=1}^{n} E[Z_{n,i}]E[(2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(por independência)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot E[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(pela equação (12.1))} \\
&= \frac{2}{n} \sum_{i=1}^{n} E[\max(Y_{i-1}, Y_{n-i})] && \text{(pela equação (C.22))} \\
&\le \frac{2}{n} \sum_{i=1}^{n} (E[Y_{i-1}] + E[Y_{n-i}]) && \text{(pelo Exercício C.3-4).}
\end{aligned}$$

Visto que cada termo $E[Y_0], E[Y_1], ..., E[Y_{n-1}]$ aparece duas vezes no último somatório, uma vez como $E[Y_{i-1}]$ e uma vez como $E[Y_{n-i}]$, temos a recorrência

$$E[Y_n] \le \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i]. \tag{12.2}$$

Usando o método de substituição, agora mostraremos que, para todos os inteiros positivos n, a recorrência (12.2) tem a solução

$$E[Y_n] \le \frac{1}{4}\binom{n+3}{3}.$$

Ao fazermos isso, usaremos a identidade

$$\sum_{i=0}^{n-1}\binom{i+3}{3} = \binom{n+3}{4}. \qquad (12.3)$$

(O Exercício 12.41 pede que você prove essa identidade.)

Para os casos-bases, observamos que os limites $0 = Y_0 = \mathrm{E}\,[Y_0] \le (1/4)\binom{3}{3} = 1/4$ e $1 = Y_1 = \mathrm{E}\,[Y_1] \le (1/4)\binom{1+3}{3} = 1$ são válidos. Para o caso indutivo, temos que

$$\begin{aligned}
E[Y_n] &\le \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i] \\
&\le \frac{4}{n}\sum_{i=0}^{n-1} \frac{1}{4}\binom{i+3}{3} && \text{(pela hipótese de indução)} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} && \text{(pela equação 12.3))} \\
&= \frac{1}{n}\cdot\frac{(n+3)!}{4!(n-1)!} \\
&= \frac{1}{4}\cdot\frac{(n+3)!}{3!n!} \\
&= \frac{1}{4}\binom{n+3}{3}.
\end{aligned}$$

Limitamos $\mathrm{E}[Y_n]$, mas nosso objetivo final é limitar $\mathrm{E}[X_n]$. Como o Exercício 12.44 pede que você mostre, a função $f(x) = 2^x$ é convexa (veja página 868). Portanto, podemos empregar a desigualdade de Jensen (C.26), que diz que

$$\begin{aligned}
2^{E[X_n]} &\le E[2^{X_n}] \\
&= E[Y_n],
\end{aligned}$$

da seguinte maneira:

$$\begin{aligned}
2^{E[X_n]} &\le \frac{1}{4}\binom{n+3}{3} \\
&= \frac{1}{4}\cdot\frac{(n+3)(n+2)(n+1)}{6} \\
&= \frac{n^3+6n^2+11n+6}{24}.
\end{aligned}$$

Tomando logaritmos de ambos os lados, temos $\mathrm{E}[X_n] = O(\lg n)$.

Exercícios

12.4-1 Prove a equação (12.3).

12.4-2 Descreva uma árvore de busca binária em n nós tal que a profundidade média de um nó na árvore seja $Q(\lg n)$, mas a altura da árvore seja $(\lg n)$. Dê um limite superior assintótico para a altura de uma árvore de busca binária de n nós na qual a profundidade média de um nó seja $Q(\lg n)$.

12.4-3 Mostre que a noção de uma árvore de busca binária escolhida aleatoriamente em n chaves, onde cada árvore de busca binária de n chaves tem igual probabilidade de ser escolhida, é diferente da noção de uma árvore de busca binária construída aleatoriamente dada nesta seção. (*Sugestão:* Faça uma lista de possibilidades quando $n = 3$.)

12.4-4 Mostre que a função $f(x) = 2^x$ é convexa.

12.4-5 ★ Considere uma aplicação de RANDOMIZED-QUICKSORT a uma sequência de n números de entrada distintos. Prove que, para qualquer constante $k > 0$, todas as $n!$ permutações de entrada exceto $O(1/n_k)$ produzem um tempo de execução $O(n \lg n)$.

Problemas

12-1 Árvores de busca binária com chaves iguais

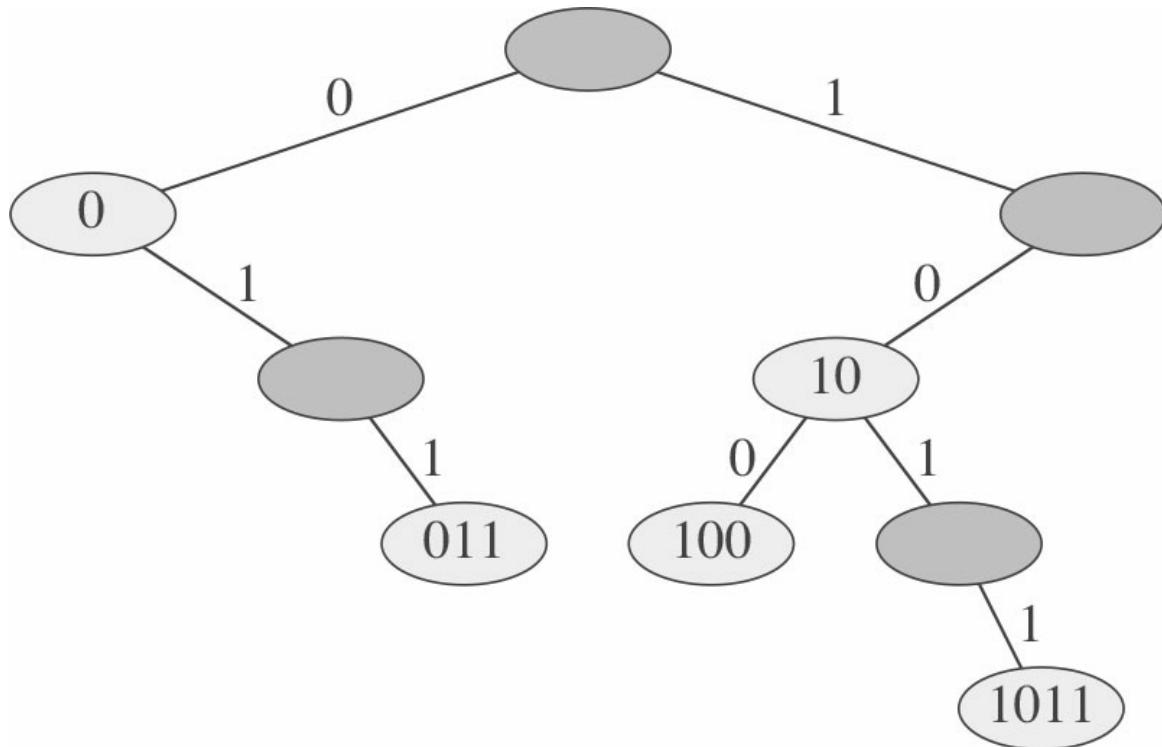
Chaves iguais apresentam um problema para a implementação de árvores de busca binária.

a. Qual é o desempenho assintótico de TREE-INSERT quando usado para inserir n itens com chaves idênticas em uma árvore de busca binária inicialmente vazia?

Propomos melhorar TREE-INSERT testando antes da linha 5 para determinar se $z.chave = x.chave$, e testando antes da linha 11 para determinar se $z.chave = y.chave$. Se a igualdade for válida, implementamos uma das estratégias a seguir. Para cada estratégia, determine o desempenho assintótico da inserção de n itens com chaves idênticas em uma árvore de busca binária inicialmente vazia. (As estratégias são descritas para a linha 5, na qual comparamos as chaves de z e x. Substitua x por y para chegar às estratégias para a linha 11.)

b. Mantenha um sinalizador booleano $x.b$ no nó x, e atribua a x o valor $x.esquerdo$ ou $x.direito$, de acordo com o valor de $x.b$, que se alterna entre FALSE e TRUE a cada vez que visitamos x durante a inserção de um nó com a mesma chave que x.

c. Mantenha em x uma lista de nós com chaves iguais e insira z na lista.

d. Defina aleatoriamente x como $x.esquerdo$ ou $x.direito$. (Dê o desempenho do pior caso e deduza informalmente o tempo de execução esperado.)

12-2 Árvores digitais

Dadas duas cadeias $a = a_0a_1 \ldots a_p$ e $b = b_0b_1 \ldots b_q$, onde cada a_i e cada b_j está em algum conjunto ordenado de caracteres, dizemos que a cadeia a é ***lexicograficamente menor que*** a cadeia b se

1. existe um inteiro j, onde $0 \le j \le \min(p, q)$, tal que $a_i = b_i$ para todo $i = 0, 1, \ldots, j - 1$ e $a_j < b_j$ ou
2. $p < q$ e $a_i = b_i$ para todo $i = 0, 1, \ldots, p$.

Por exemplo, se a e b são cadeias de bits, então 10100 < 10110 pela regra 1 (fazendo $j = 3$) e 10100 < 101000 pela regra 2. Essa ordenação é semelhante à utilizada em dicionários de idiomas.

A estrutura de dados ***árvore digital*** mostrada na Figura 12.5 armazena as cadeias de bits 1011, 10, 011, 100 e 0. Quando procuramos uma chave a $a = a_0a_1 \ldots a_p$, vamos para a esquerda em um nó de profundidade i se $a_i = 0$ e para a direita se $a_i = 1$. Seja S um conjunto de cadeias binárias distintas cujos comprimentos somam n. Mostre como usar uma árvore digital para ordenar S lexicograficamente no tempo $Q(n)$. No exemplo da Figura 12.6, a saída da ordenação deve ser a sequência 0, 011, 10, 100, 1011.

Figura 12.5 Uma árvore digital que armazena as cadeias de bits 1011, 10, 011, 100 e 0. Podemos determinar a chave de cada nó percorrendo o caminho simples da raiz até esse nó. Portanto, não há necessidade de armazenar as chaves nos nós: as chaves são mostradas aqui somente para fins ilustrativos. Os nós estão sombreados em tom mais escuro se as chaves correspondentes a eles não estão na árvore; esses nós estão presentes apenas para estabelecer um caminho até outros nós.

12-3 Profundidade média de nó em uma árvore de busca binária construída aleatoriamente

Neste problema, provamos que a profundidade média de um nó em uma árvore de busca binária construída aleatoriamente com n nós é $O(\lg n)$. Embora esse resultado seja mais fraco que o do Teorema 12.4, a técnica que empregaremos revelará uma semelhança surpreendente entre a construção de uma árvore de busca binária e a execução do algoritmo RANDOMIZED-QUICKSORT da Seção 7.3.

Definimos o ***comprimento total de caminho*** $P(T)$ de uma árvore binária T como a soma, contando todos os nós x em T, da profundidade do nó x, que denotamos por $d(x, T)$.

a. Demonstre que a profundidade média de um nó em T é

$$\frac{1}{n}\sum_{x \in T} d(x,T) = \frac{1}{n}P(T).$$

Assim, desejamos mostrar que o valor esperado de $P(T)$ é $O(n \lg n)$.

b. Denotemos por T_E e T_D as subárvores esquerda e direita da árvore T, respectivamente. Mostre que, se T tem n nós, então

$$P(T) = P(T_E) + P(T_D) + n - 1 .$$

c. Seja $P(n)$ o comprimento total de caminho médio de uma árvore de busca binária construída aleatoriamente com n nós. Mostre que

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1).$$

d. Mostre que $P(n)$ pode ser reescrito como

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1}P(k) + \Theta(n).$$

e. Recordando a análise alternativa da versão aleatorizada do quicksort dada no Problema 73, conclua que $P(n) = O(n \lg n)$.

A cada invocação recursiva do quicksort, escolhemos um elemento pivô aleatório para particionar o conjunto de elementos que está sendo ordenado. Cada nó de uma árvore de busca binária particiona o conjunto de elementos que cai na subárvore digital nesse nó.

f. Descreva uma implementação do quicksort na qual as comparações para ordenar um conjunto de elementos são exatamente iguais às comparações para inserir os elementos em uma árvore de busca binária. (A ordem em que as comparações são efetuadas pode ser diferente, mas as mesmas comparações devem ser executadas.)

12-4 Número de árvores binárias diferentes

Seja b_n o número de árvores binárias diferentes com n nós. Neste problema, você determinará uma fórmula para b_n, bem como uma estimativa assintótica.

a. Mostre que $b_0 = 1$ e que, para $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$

b. Consultando o Problema 44 para a definição de uma função geradora, seja $B(x)$ a função geradora

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

Mostre que $B(x) = xB(x)^2 + 1$ e, consequentemente, um modo de expressar $B(x)$ em forma fechada é

$$B(x) = \frac{1}{2x}(1 - \sqrt{1-4x}).$$

A ***expansão de Taylor*** de $f(x)$ em torno do ponto $x = a$ é dada por

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k .$$

onde $f^{(k)}(x)$ é a $k$ésima derivada de f avaliada em x.

c. Mostre que

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(o $n$ésimo ***número de Catalan***) usando a expansão de Taylor de $\sqrt{1-4x}$ m torno de $x = 0$. (Se desejar, em vez de usar a expansão de Taylor, você pode empregar a generalização da expansão binomial (C.4) para expoentes não inteiros n onde, para qualquer número real n e qualquer inteiro k, interpretamos $\binom{n}{k}$ como $n(n-1) \ldots (n-k+1)/k!$ se $k \geq 0$, e 0 em caso contrário.)

d. Mostre que

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}}(1 + O(1/n)).$$

NOTAS DO CAPÍTULO

Knuth [211] contém uma boa discussão de árvores de busca binária simples, bem como de muitas variações. Parece que as árvores de busca binária foram descobertas independentemente por várias pessoas no final da década de 1950. Árvores digitais são frequentemente denominadas "tries", que vem das letras do meio da palavra *retrieval* em inglês, que significa "recobrar" ou "reaver". Knuth também as discutiu [211].

Muitos textos, entre eles as duas primeiras edições deste livro, apresentam um método um pouco mais simples de eliminar um nó de uma árvore de busca binária quando ambos os seus filhos estão presentes. Em vez de substituir o nó z por seu sucessor y, eliminamos o nó y mas copiamos sua chave e os dados satélites para o nó z. A desvantagem dessa abordagem é que o nó realmente eliminado poderia não ser o nó passado para o procedimento de eliminação. Se outros componentes de um programa mantiverem ponteiros para nós na árvore, poderiam acabar erroneamente com ponteiros "vencidos" para nós que já foram eliminados. Embora seja um pouco mais complicado, o método de eliminação apresentado nesta edição deste livro garante que uma chamada para eliminar o nó z elimina o nó z e somente o nó z.

A Seção 15.5 mostrará como construir uma árvore de busca binária ótima quando conhecemos as frequências de busca antes da construção da árvore. Isto é, dadas as frequências de busca para cada chave e as frequências de busca para valores que caem entre chaves na árvore, construímos uma árvore de busca binária para a qual um conjunto de buscas que decorre dessas frequências examina o número mínimo de nós.

A prova da Seção 12.4 que limita a altura esperada de uma árvore de busca binária construída aleatoriamente se deve a Aslam [24]. Martínez e Roura [243] dão algoritmos aleatorizados para inserção e eliminação em árvores de busca binária nos quais o resultado de qualquer dessas operações é uma árvore de busca binária aleatória. Contudo, a definição desses autores para uma árvore de busca binária é diferente — apenas ligeiramente — da definição de uma árvore de busca binária construída aleatoriamente dada neste capítulo.

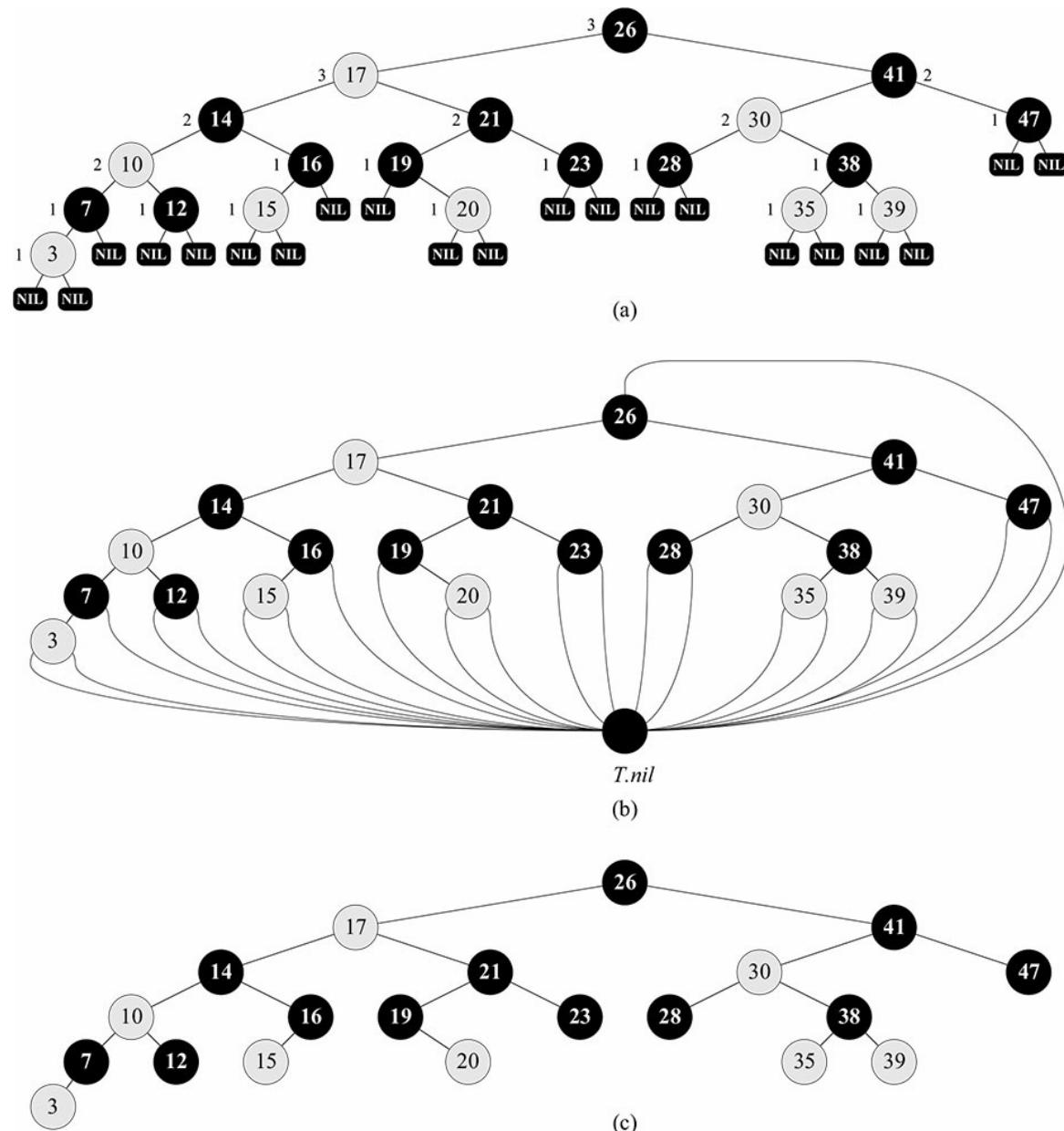
13 ÁRVORES VERMELHO-PRETO

O Capítulo 12 mostrou que uma árvore de busca binária de altura h pode suportar qualquer das operações básicas de conjuntos dinâmicos — como SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT e DELETE — no tempo $O(h)$. Assim, as operações de conjuntos são rápidas se a altura da árvore de busca é pequena. Todavia, se a altura da árvore é grande, a execução dessas operações poderá ser mais lenta do que com uma lista ligada. Árvores vermelho-preto são um dos muitos esquemas de árvores de busca que são "balanceadas" de modo a garantir que operações básicas de conjuntos dinâmicos demorem o tempo $O(\lg n)$ no pior caso.

13.1 PROPRIEDADES DE ÁRVORES VERMELHO-PRETO

Uma ***árvore vermelho-preto*** é uma árvore de busca binária com um bit extra de armazenamento por nó: sua ***cor*** — ela pode ser VERMELHA ou PRETA. Restringindo as cores dos nós em qualquer caminho simples da raiz até uma folha, as árvores vermelho-preto asseguram que o comprimento de nenhum desses caminhos seja maior que duas vezes o de qualquer outro, de modo que a árvore é aproximadamente ***balanceada***.

Cada nó da árvore contém agora os atributos *cor*, *chave*, *esquerda*, *direita* e *p*. Se um filho ou o pai de um nó não existir, o atributo do ponteiro correspondente do nó contém o valor NIL. Trataremos esses valores NIL como se fossem ponteiros para folhas (nós externos) da árvore de busca binária e os nós normais que portam chaves como nós internos da árvore.

Uma árvore vermelho-preto é uma árvore de busca binária que satisfaz as seguintes ***propriedades vermelho-preto***:

1. Todo nó é vermelho ou preto.
2. A raiz é preta.
3. Toda folha (NIL) é preta.
4. Se um nó é vermelho, então os seus filhos são pretos.
5. Para cada nó, todos os caminhos simples do nó até folhas descendentes contêm o mesmo número de nós pretos.

A Figura 13.1 mostra um exemplo de árvore vermelho-preto.

Por questão de conveniência no tratamento das condições de fronteira em código de árvores vermelho-preto, usamos uma única sentinela para representar NIL (veja p. 238). Para uma árvore vermelho-preto T, a sentinela $T.nil$ é um objeto com os mesmos atributos que um nó comum na árvore. Seu atributo *cor* é PRETO e seus outros atributos — *p*, *esquerda*, *direita* e *chave* — podem adotar valores arbitrários. Como mostra a Figura 13.1(b), todos os ponteiros para NIL são substituídos por ponteiros para a sentinela $T.nil$.

Usamos a sentinela para poder tratar um filho NIL de um nó x como um nó comum cujo pai é x. Se bem que poderíamos adicionar um nó de sentinela distinto para cada NIL na árvore, de modo que o pai de cada NIL fosse bem definido, essa abordagem desperdiçaria espaço. Em vez disso, usamos a única sentinela $T.nil$ para representar todos os

nós NIL — todas as folhas e o pai da raiz. Os valores dos atributos *p*, *esquerda*, *direita* e *chave* da sentinela são irrelevantes, embora, por conveniência, possamos defini-los durante o curso de um procedimento.

Figura 13.1 Uma árvore vermelho-preto com nós pretos em negro e nós vermelhos em cinzento. Todo nó em uma árvore vermelho-preto é vermelho ou preto, os filhos de um nó vermelho são pretos, e todo caminho simples de um nó até uma folha descendente contém o mesmo número de nós pretos. **(a)** Toda folha, mostrada como um NIL, é preta. Cada nó não NIL é marcado com sua altura preta: nós NILs têm altura preta igual a 0. **(b)** A mesma árvore vermelho-preto, mas com cada NIL substituído pela única sentinela *T.nil* , que é sempre preta, e cujas alturas pretas são omitidas. O pai da raiz também é a sentinela. **(c)** A mesma árvore vermelho-preto, mas com folhas e o pai da raiz omitidos completamente. Utilizaremos esse estilo de representação no restante deste capítulo.

Em geral, limitamos nosso interesse aos nós internos de uma árvore vermelho-preto, já que eles contêm os valores de chaves. No restante deste capítulo, omitiremos as folhas quando desenharmos árvores vermelho-preto, como mostra a Figura 13.1(c).

Denominamos o número de nós pretos em qualquer caminho simples de um nó x, sem incluir esse nó, até uma folha, por ***altura preta*** do nó, denotada por bh(x). Pela propriedade 5, a noção de altura preta é bem definida, já que

todos os caminhos simples descendentes que partem do nó têm o mesmo número de nós pretos. Definimos a altura preta de uma árvore vermelho-preto como a altura preta de sua raiz.

O lema a seguir, mostra por que as árvores vermelho-preto dão boas árvores de busca.

Lema 13.1

Uma árvore vermelho-preto com n nós internos tem, no máximo, a altura $2 \lg(n + 1)$.

Prova Começamos mostrando que a subárvore com raiz em qualquer nó x contém no mínimo $2^{bh(x)} - 1$ nós internos. Provamos essa afirmativa por indução sobre a altura de x. Se a altura de x é 0, então x deve ser uma folha ($T.nil$), e a subárvore com raiz em x realmente contém no mínimo $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nós internos. Para a etapa indutiva, considere um nó x que tenha altura positiva e considere um nó interno x com dois filhos. Cada filho tem uma altura preta $bh(x)$ ou $bh(x) - 1$, dependendo de sua cor ser vermelha ou preta, respectivamente. Visto que a altura de um filho de x é menor que a altura do próprio x, podemos aplicar a hipótese indutiva para concluir que cada filho tem, no mínimo, $2^{bh(x) - 1} - 1$ nós internos. Assim, a subárvore com raiz em x contém, no mínimo, $(2^{bh(x) - 1} - 1) + (2^{bh(x) - 1} - 1) + 1 = 2^{bh(x)} - 1$ nós internos, o que prova a afirmativa.

Para completar a prova do lema, seja h a altura da árvore. De acordo com a propriedade 4, no mínimo metade dos nós em qualquer caminho simples da raiz até uma folha, não incluindo a raiz, deve ser preta. Consequentemente, a altura preta da raiz deve ser, no mínimo, $h/2$; assim,

$$n \geq 2^{h/2} - 1 .$$

Passando o valor 1 para o lado esquerdo e tomando logaritmos em ambos os lados, temos $\lg(n + 1) \geq h/2$ ou $h \leq 2 \lg(n + 1)$.

Uma consequência imediata desse lema é que podemos implementar as operações de conjuntos dinâmicos SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR no tempo $O(\lg n)$ em árvores vermelho-preto, já que cada execução no tempo $O(h)$ em uma árvore de busca de altura h (como mostra o Capítulo 12) e em qualquer árvore vermelho-preto em n nós é uma árvore de busca com altura $O(\lg n)$. (Claro que as referências a NIL nos algoritmos do Capítulo 12 teriam de ser substituídas por $T.nil$.) Embora os algoritmos TREE-INSERT e TREE-DELETE do Capítulo 12 sejam executados no tempo $O(\lg n)$ quando é dada uma árvore vermelho-preto como entrada, eles não suportam diretamente as operações de conjuntos dinâmicos INSERT e DELETE, já que não garantem que a árvore de busca binária modificada será uma árvore vermelho-preto. Porém, veremos nas Seções 13.3 e 13.4 como suportar essas duas operações no tempo $O(\lg n)$.

Exercícios

13.1-1 Desenhe, no estilo da Figura 13.1(a) , a árvore de busca binária completa de altura 3 nas chaves {1, 2, ..., 15}. Adicione as folhas NIL e dê três cores diferentes aos nós, de tal modo que as alturas pretas das árvores vermelho-preto resultantes sejam 2, 3 e 4.

13.1-2 Desenhe a árvore vermelho-preto que resulta após a chamada a TREE-INSERT na árvore da Figura 13.1 com chave 36. Se o nó inserido for vermelho, a árvore resultante é uma árvore vermelho-preto? E se ele for preto?

13.1-3 Vamos definir uma ***árvore vermelho-preto relaxada*** como uma árvore de busca binária que satisfaz as propriedades vermelho-preto 1, 3, 4 e 5. Em outras palavras, a raiz pode ser vermelha ou preta. Considere uma árvore vermelho-preto relaxada T cuja raiz é vermelha. Se colorirmos a raiz de T de preto, mas não fizermos nenhuma outra mudança em T, a árvore resultante é uma árvore vermelho-preto?

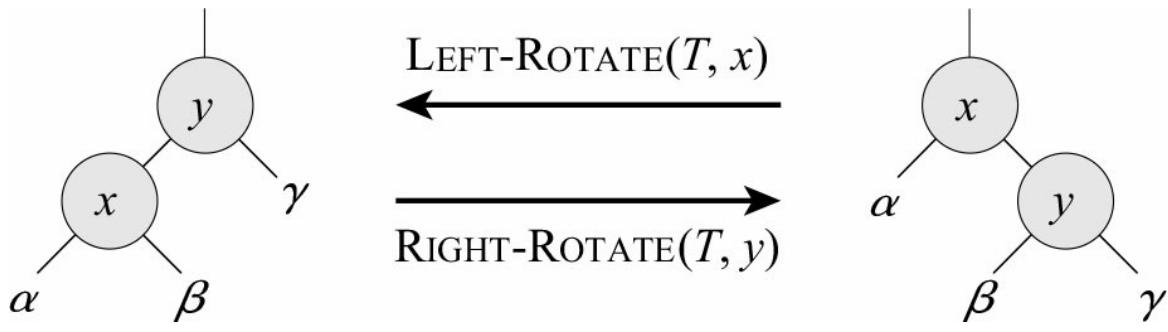
13.1-4 Suponha que "absorvemos" todo nó vermelho em uma árvore vermelho-preto em seu pai preto, de modo que os filhos do nó vermelho se tornem filhos do pai preto. (Ignore o que acontece com as chaves.) Quais são os

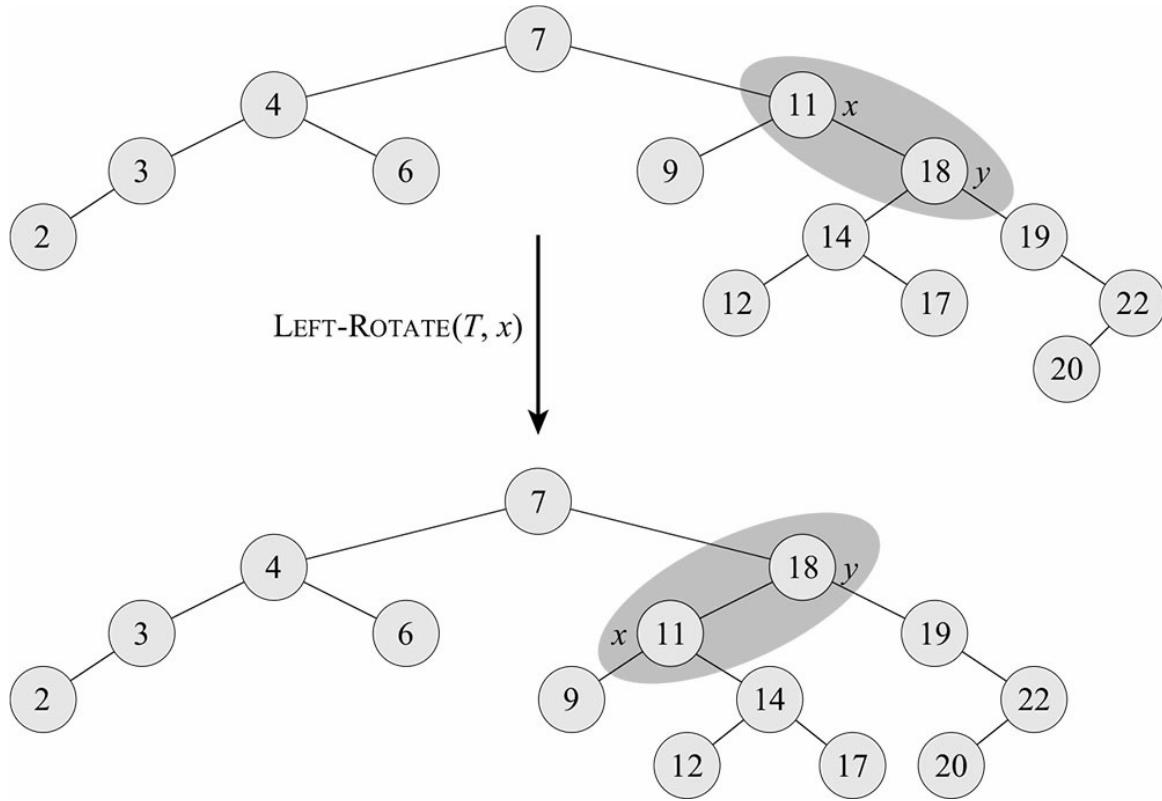
graus possíveis de um nó preto depois que todos os seus filhos vermelhos são absorvidos? O que você pode dizer sobre as profundidades das folhas da árvore resultante?

13.1-5 Mostre que o comprimento do mais longo caminho simples de um nó x em uma árvore vermelho-preto até uma folha descendente é, no máximo, duas vezes o do caminho simples mais curto do nó x até uma folha descendente.

13.1-6 Qual é o maior número possível de nós internos em uma árvore vermelho-preto com altura preta k? Qual é o menor número possível?

13.1-7 Descreva uma árvore vermelho-preto em n chaves que permita a maior razão possível entre nós internos vermelhos e nós internos pretos. Qual é essa razão? Qual árvore tem a menor razão possível e qual é essa razão?

13.2 Rotações

As operações de árvores de busca Tree-Insert e Tree-Delete, quando executadas em uma árvore vermelho-preto com n chaves, demoram o tempo $O(\lg n)$. Como elas modificam a árvore, o resultado pode violar as propriedades vermelho-preto enumeradas na Seção 13.1. Para restabelecer essas propriedades, devemos mudar as cores de alguns nós na árvore e também mudar a estrutura de ponteiros.

Mudamos a estrutura de ponteiros por meio de ***rotação***, uma operação local em uma árvore de busca que preserva a propriedade de árvore de busca binária. A Figura 13.2 mostra os dois tipos de rotações: rotações para a esquerda e rotações para a direita. Quando fazemos uma rotação para a esquerda em um nó x, supomos que seu filho à direita y não é *T.nil*; x pode ser qualquer nó na árvore cujo filho à direita não é *T.nil*. A rotação para a esquerda "pivota" ao redor da ligação de x para y. Transforma y na nova raiz da subárvore, com x como filho à esquerda de y e o filho à esquerda de y como filho à direita de x.

O pseudocódigo para Left-Rotate supõe que $x.direita \neq T.nil$ e que o pai da raiz é *T.nil*.

```
LEFT-ROTATE(T; x)
1  y = x.direita            // define y
2  x.direita = y.esquerda   // transforma a subárvore à esquerda de y na subárvore à direita de x
3  if y.esquerda ≠ T.nil
4        y.esquerda.p = x
5  y.p = x.p                // liga o pai de x a y
6  if x.p == T.nil
7        T.raiz = y
8  elseif x == x.p.esquerda
9        x.p. esquerda = y
10 else x.p.direita = y
11 y.esquerda = x           // coloca x à esquerda de y
12 x.p = y
```

A Figura 13.3 mostra um exemplo de como Left-Rotate modifica uma árvore de busca binária. O código para Right-Rotate é simétrico. Left-rotate e Right-Rotate são executados no tempo $O(1)$. Somente ponteiros são alterados por uma rotação; todos os outros atributos em um nó permanecem os mesmos.

Figura 13.2 As operações de rotação em uma árvore de busca binária. A operação LEFT-ROTATE(T, x) transforma a configuração dos dois nós à direita na configuração à esquerda mudando um número constante de ponteiros. A operação inversa RIGHT-ROTATE(T, y) transforma a configuração à esquerda na configuração à direita. As letras α, β e g representam subárvores arbitrárias. Uma operação de rotação preserva a propriedade de árvore de busca binária: as chaves em α precedem $x.chave$, que precede as chaves em β, que precedem $y.chave$, que precede as chaves em g.

Figura 13.3 Um exemplo de como o procedimento LEFT-ROTATE(T, x) modifica uma árvore de busca binária. Os percursos de árvore em in-ordem da árvore de entrada e a árvore modificada produzem a mesma listagem de valores de chaves.

Exercícios

13.2-1 Escreva pseudocódigo para RIGHR-ROTATE.

13.2-2 Demonstre que, em toda árvore de busca binária de n nós, existem exatamente $n - 1$ rotações possíveis.

13.2-3 Sejam a, b e c nós arbitrários nas subárvores α, β e γ, respectivamente, na árvore da direita da Figura 13.2. Como as profundidades de a, b e c mudam quando é realizada uma rotação para a esquerda no nó x na

figura?

13.2-4 Mostre que qualquer árvore de busca binária arbitrária de n nós pode ser transformada em qualquer outra árvore de busca binária arbitrária de n nós por meio de $O(n)$ rotações. (*Sugestão*: Primeiro, mostre que, no máximo, $n - 1$ rotações para a direita são suficientes para transformar a árvore em uma cadeia orientada para a direita.)

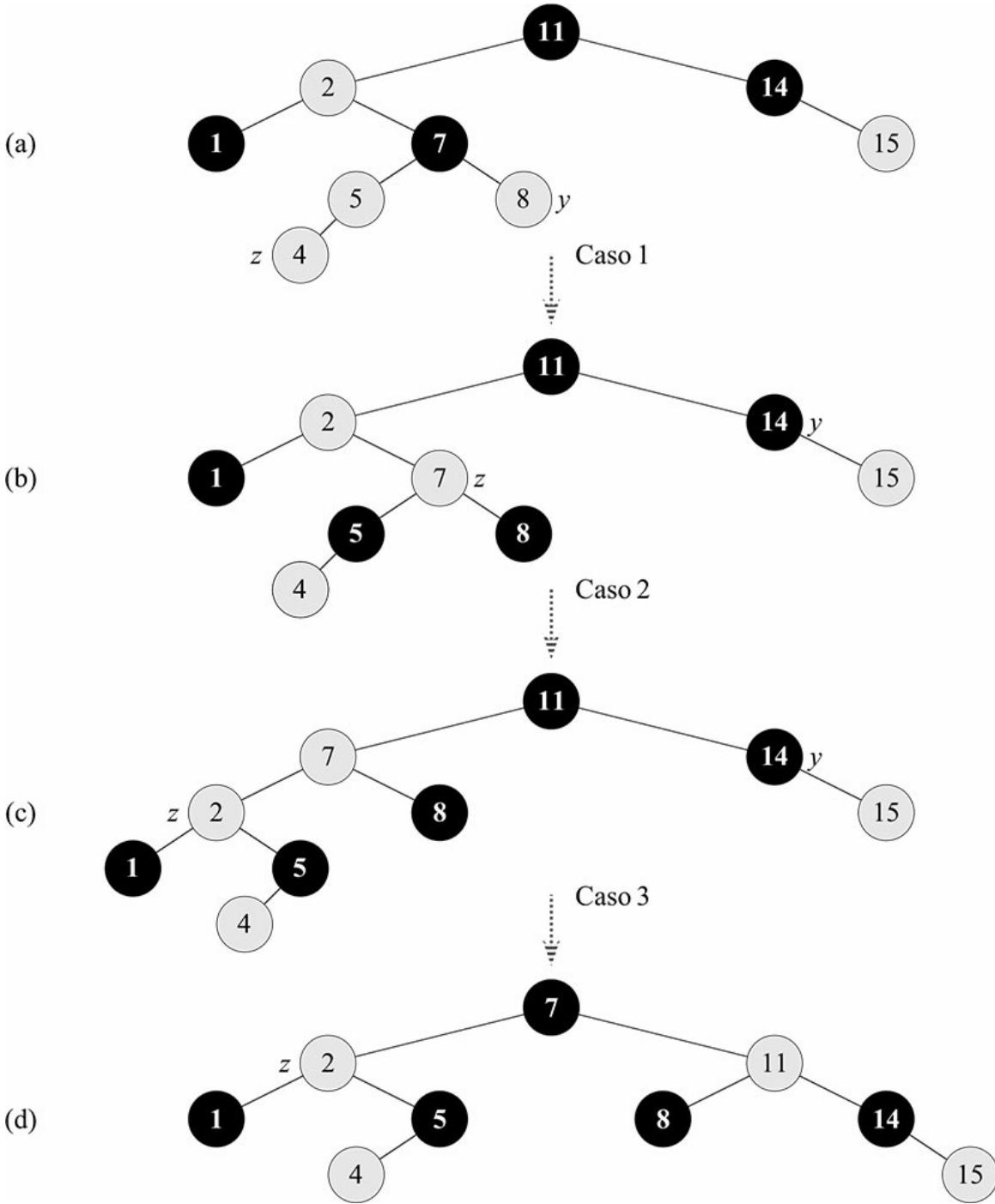
13.2-5 ★

Dizemos que uma árvore de busca binária T_1 pode ser ***convertida para a direita*** na árvore de busca binária T_2 se for possível obter T_2 de T_1 por meio de uma série de chamadas a RIGHR-ROTATE. Dê um exemplo de duas árvores T_1 e T_2 tais que T_1 não possa ser convertida para a direita em T_2. Em seguida, mostre que, se uma árvore T_1 pode ser convertida para a direita em T_2, ela pode ser convertida para a direita por meio de $O(n_2)$ chamadas a RIGHT-ROTATE.

13.3 INSERÇÃO

Podemos inserir um nó em uma árvore vermelho-preto de n nós no tempo $O(\lg n)$. Para tal, usamos uma versão ligeiramente modificada do procedimento TREE-INSERT (Seção 12.3) para inserir o nó z na árvore T como se ela fosse uma árvore de busca binária comum e depois colorimos z de vermelho. (O Exercício 13.3-1 pede que você explique por que escolhemos que o nó z é vermelho, em vez de preto.) Para garantir que as propriedades vermelho-preto serão preservadas, chamamos um procedimento auxiliar RB-INSERT-FIXUP para colorir novamente os nós e executar rotações. A chamada RB-INSERT(T, z) insere o nó z — cuja *chave* considera-se já ter sido inserida — na árvore vermelho-preto T.

```
RB-INSERT(T,z)
1    y = T.nil
2    x = T.raiz
3    while x ≠ T.nil
4        y = x
5        if z.chave < x.chave
6            x = x.esquerda
7        else x = x.direita
8    z.p = y
9    if y == T.nil
10       T.raiz = z
11   elseif z.chave < x.chave
12       y.esquerda = z
13   else y.direita = z
14   z.esquerda = T.nil
15   z.direita = T.nil
16   z.cor = RED
17   RB-INSERT-FIXUP(T,z)
```

Há quatro diferenças entre os procedimentos TREE-INSERT e RB-INSERT. Primeiro, todas as instâncias de NIL em TREE-INSERT são substituídas por *T.nil*. Em segundo lugar, definimos *z.esquerda* e *z.direita* como *T.nil* nas linhas 14 e 15 de RB-INSERT, a fim de manter a estrutura de árvore adequada. Em terceiro lugar, colorimos z de vermelho na linha 16. Em quarto lugar, visto que colorir z de vermelho pode causar uma violação de uma das propriedades vermelho-preto, chamamos RB-INSERT-FIXUP(T, z) na linha 17 de RB-INSERT para restaurar as propriedades vermelho-preto.

```
RB-INSERT-FIXUP(T, z)
1    while z.p.cor == VERMELHO
2        if z.p == z.p.p.esquerda
3            y = z.p.p.direita
4            if y.cor = VERMELHO
5                z.p.cor = PRETO                     // caso 1
6                y.cor = PRETO                       // caso 1
7                z.p.p.cor = VERMELHO                // caso 1
8                z = z.p.p                           // caso 1
9            else if z = z.p.direita
10                   z = z.p                         // caso 2
11                   LEFT-ROTATE(T, z)               // caso 2
12               z.p.cor = PRETO                     // caso 3
13               z.p.p.cor = VERMELHO                // caso 3
14               RIGHT-ROTATE(T, z.p.p)              // caso 3
15       else (igual à cláusula then
                 com "direita" e "esquerda" trocadas)
16   T.raiz.cor = PRETO
```

Para entender como RB-INSERT-FIXUP funciona, desmembraremos nosso exame do código em três etapas principais. Primeiro, determinaremos quais violações das propriedades vermelho-preto são introduzidas em RB-INSERT quando o nó *z* é inserido e colorido de vermelho. Em segundo lugar, examinaremos a meta global do laço **while** das linhas 1−15. Por fim, exploraremos cada um dos três casos[1] dentro do corpo do laço **while** e veremos como eles cumprem essa meta. A Figura 13.4 mostra como RB-INSERT-FIXUP funciona em uma amostra de árvore vermelho-preto.

Quais das propriedades vermelho-preto podem ser violadas na chamada a RB-INSERT-FIXUP? A propriedade 1 certamente continua válida, bem como a propriedade 3, já que ambos os filhos do nó vermelho recém-inserido são a sentinela *T.nil*. A propriedade 5, que diz que o número de nós pretos é igual em todo caminho simples de um dado nó, também é satisfeita porque o nó *z* substitui a sentinela (preta), e o nó *z* é vermelho com filhos sentinelas. Assim, as únicas propriedades que poderiam ser violadas são a propriedade 2, que exige que a raiz seja preta, e a propriedade 4, que diz que um nó vermelho não pode ter um filho vermelho. Ambas as violações possíveis se devem a *z* ser colorido de vermelho. A propriedade 2 é violada se *z* é a raiz, e a propriedade 4 é violada se o pai de *z* é vermelho. A Figura 13.4(a) mostra uma violação da propriedade 4 após a inserção do nó *z*.

Figura 13.4 A operação de RB-INSERT-FIXUP. **(a)** Um nó z depois da inserção. Como z e seu pai $z.p$ são vermelhos, ocorre uma violação da propriedade 4. Visto que o tio y de z é vermelho, o caso 1 no código se aplica. Colorimos novamente os nós e movimentamos o ponteiro z para cima na árvore, resultando na árvore mostrada em **(b)**. Mais uma vez, z e seu pai são vermelhos, mas o tio y de z é preto. Como z é o filho à direita de $z.p$, o caso 2 se aplica. Executamos uma rotação para a esquerda e a árvore resultante é mostrada em **(c)**. Agora, z é o filho à esquerda de seu pai, e o caso 3 se aplica. Colorindo novamente e executando uma rotação para a direita, é produzida a árvore em **(d)**, que é uma árvore vermelho-preto válida.

O laço **while** nas linhas 1−15 mantém o seguinte invariante de três partes no início de cada iteração do laço:

a. O nó z é vermelho.
b. Se $z.p$ é a raiz, então $z.p$ é preto.

c. Se a árvore violar qualquer das propriedades vermelho-preto, ela violará no máximo uma delas, e a violação será da propriedade 2 ou da propriedade 4. Se a árvore violar a propriedade 2 é porque *z* é a raiz e é vermelho. Se a árvore violar a propriedade 4 é porque *z* e *z.p* são vermelhos.

A parte (c), que trata das violações de propriedades vermelho-preto, é mais fundamental para mostrar que RB-Insert-Fixup restaura as propriedades vermelho-preto que as partes (a) e (b), que utilizamos no caminho para entender situações no código. Como nos concentraremos no nó *z* e nós próximos a ele na árvore, é útil saber pela parte (a) que *z* é vermelho. Usaremos a parte (b) para mostrar que o nó *z.p.p* existe quando nos referimos a ele nas linhas 2, 3, 7, 8, 13 e 14.

Lembre-se de que precisamos mostrar que um invariante de laço é verdadeiro antes da primeira iteração do laço, que cada iteração mantém o invariante de laço e que o invariante de laço nos dá uma propriedade útil ao término do laço.

Começamos com os argumentos de inicialização e término. Então, à medida que examinarmos com mais detalhes como o corpo do laço funciona, demonstraremos que o laço mantém o invariante em cada iteração. Durante o processo, também demonstraremos que cada iteração do laço tem dois resultados possíveis: o ponteiro *z* sobe a árvore ou executamos algumas rotações e o laço termina.

Inicialização: Antes da primeira iteração do laço, começamos com uma árvore vermelho-preto sem nenhuma violação e acrescentamos um nó vermelho *z*. Mostramos que cada parte do invariante é válida no momento em que RB-Insert-Fixup é chamado:

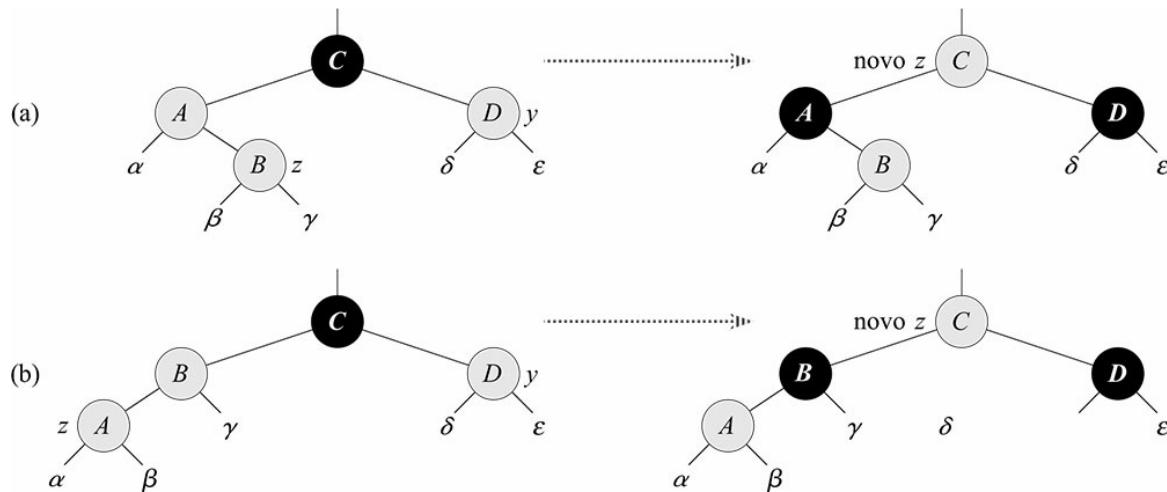
a. Quando RB-Insert-Fixup é chamado, *z* é o nó vermelho que foi acrescentado.

b. Se *p*[*z*] é a raiz, então *z.p* começou preto e não mudou antes da chamada de RB--Insert-Fixup.

c. Já vimos que as propriedades 1, 3 e 5 são válidas quando RB-Insert-Fixup é chamado. Se a árvore violar a propriedade 2, a raiz vermelha deve ser o nó *z* recém-acrescentado, que é o único nó interno na árvore. Como o pai e ambos os filhos de *z* são a sentinela, que é preta, a árvore tampouco viola a propriedade 4. Assim, essa violação da propriedade 2 é a única violação de propriedades vermelho-preto na árvore inteira. Se a árvore violar a propriedade 4, como os filhos do nó *z* são sentinelas pretas e a árvore não tinha nenhuma outra violação antes de *z* ser acrescentado, a violação tem de ser porque *z* e *z.p* são vermelhos. Além disso, a árvore não viola nenhuma outra propriedade vermelho-preto.

Término: Quando o laço termina, é porque *z.p* é preto. (Se *z* é a raiz, então *z.p* é a sentinela *T.nil*, que é preta.) Assim, a árvore não viola a propriedade 4 no término do laço. Pelo invariante de laço, a única propriedade que poderia deixar de ser válida é a propriedade 2. A linha 16 restaura também essa propriedade, de modo que, quando RB-Insert-Fixup termina, todas as propriedades vermelho-preto são válidas.

Manutenção: Na realidade, precisamos considerar seis casos no laço **while**, mas três deles são simétricos aos outros três, dependendo de a linha 2 determinar que o pai *z.p* de *z* é um filho à esquerda ou um filho à direita do avô *z.p.p* de *z*. Damos o código somente para a situação na qual *z.p* é um filho à esquerda. O nó *z.p.p* existe, já que, pela parte (b) do invariante de laço, se *z.p* é a raiz, então *z.p* é preto. Visto que entramos em uma iteração de laço somente se *z.p* é vermelho, sabemos que *z.p* não pode ser a raiz. Consequentemente, *z.p.p* existe.

Distinguimos o caso 1 dos casos 2 e 3 pela cor do irmão do pai de *z*, ou "tio". A linha 3 faz *y* apontar para o tio *z.p.p.direita* de *z*, e a linha 4 testa a cor de *y*. Se *y* é vermelho, então executamos o caso 1. Do contrário, o controle passa para os casos 2 e 3. Em todos os três casos, o avô *z.p.p* de *z* é preto, já que seu pai *z.p* é vermelho, e a propriedade 3 é violada apenas entre *z* e *z.p*.

Caso 1: o tio de *y* de *z* é vermelho

A Figura 13.5 mostra a situação para o caso 1 (linhas 5−8), que ocorre quando *z.p* e *y* são vermelhos. Como *z.p.p* é preto, podemos colorir *z.p* e *y* de preto, o que corrige o problema de *z* e *z.p* serem vermelhos, e podemos colorir *z.p.p* de vermelho, mantendo assim a propriedade 5. Então repetimos o laço **while** com *z.p.p* como o novo nó *z*. O ponteiro *z* sobe dois níveis na árvore. Agora mostramos que o caso 1 mantém o invariante de laço no início da próxima iteração. Usamos *z* para denotar o nó *z* na iteração atual, e *z'* = *z.p.p* para denotar o nó que será denominado *z* no teste da linha 1 na iteração seguinte.

a. Como essa iteração colore *z.p.p* de vermelho, o nó *z'* é vermelho no início da próxima iteração.
b. O nó *z'.p* é *z.p.p.p* nessa iteração, e a cor desse nó não se altera. Se esse nó é a raiz, ele era preto antes dessa iteração e permanece preto no início da próxima iteração.
c. Já mostramos que o caso 1 mantém a propriedade 5 e não introduz uma violação das propriedades 1 ou 3.

Se o nó *z'* é a raiz no início da próxima iteração, então o caso 1 corrigiu a única violação da propriedade 4 nessa iteração. Como *z'* é vermelho e é a raiz, a propriedade 2 passa a ser a única violada, e essa violação se deve a *z'*.

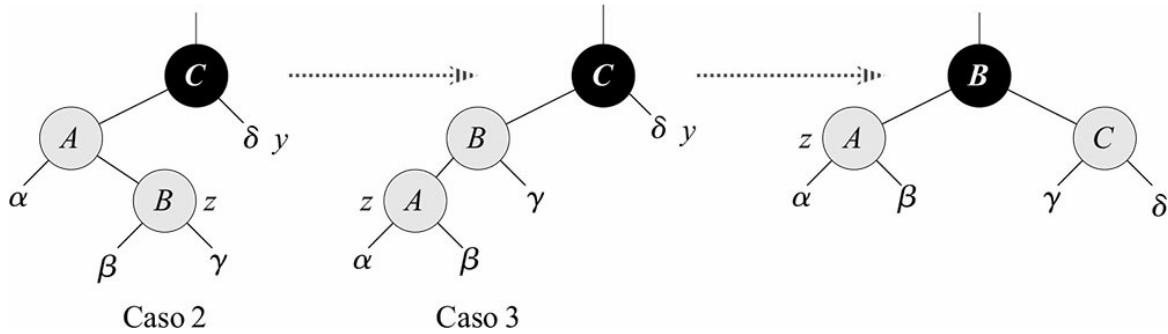
Se o nó *z'* não é a raiz no início da próxima iteração, então o caso 1 não criou uma violação da propriedade 2. O caso 1 corrigiu a única violação da propriedade 4 que existia no início dessa iteração. Então, transformou *z'* em vermelho e deixou *z'.p* como estava. Se *z'.p* era preto, não há nenhuma violação da propriedade 4. Se *z'.p* era vermelho, colorir *z'* de vermelho criou uma violação da propriedade 4 entre *z'* e *z'.p* .

Figura 13.5 O caso 1 do procedimento RB-INSERT-FIXUP. A propriedade 4 é violada, já que *z* e seu pai *z.p* são vermelhos. A mesma ação é adotada se **(a)** *z* é um filho à direita ou **(b)** *z* é um filho à esquerda. Cada uma das subárvores, α, β, *g*, *d* e *e* tem uma raiz preta e cada uma tem a mesma altura preta. O código para o caso 1 muda as cores de alguns nós, preservando a propriedade 5: todos os caminhos simples descendentes de um nó até uma folha têm o mesmo número de pretos. O laço **while** continua com o avô *z.p.p* do nó *z* como o novo *z*. Qualquer violação da propriedade 4 só pode ocorrer agora entre o novo *z*, que é vermelho, e seu pai, que também é vermelho.

Caso 2: o tio *y* de *z* é preto e *z* é um filho à direita

Caso 3: o tio *y* de *z* é preto e *z* é um filho à esquerda

Nos casos 2 e 3, a cor do tio *y* de *z* é preta. Distinguimos os dois casos conforme *z* seja um filho à direita ou à esquerda de *z.p*. As linhas 10 e 11 constituem o caso 2, que é mostrado na Figura 13.6, juntamente com o caso 3. No caso 2, o nó *z* é um filho à direita de seu pai. Usamos imediatamente uma rotação para a esquerda para transformar a situação no

caso 3 (linhas 12−14), na qual o nó z é um filho à esquerda. Como z e $z.p$ são vermelhos, a rotação não afeta a altura preta dos nós nem a propriedade 5. Quer entremos no caso 3 diretamente ou por meio do caso 2, o tio y de z é preto, já que, do contrário, teríamos executado o caso 1. Além disso, o nó $z.p.p$ existe, visto que demonstramos que esse nó existia no momento em que as linhas 2 e 3 foram executadas e, após z subir um nível na linha 10 e depois descer um nível na linha 11, a identidade de $z.p.p$ permanece inalterada. No caso 3, executamos algumas mudanças de cores e uma rotação para a direita, o que preserva a propriedade 5; em seguida, visto que não temos mais dois nós vermelhos em uma linha, encerramos. O corpo do laço **while** não é executado outra vez, já que agora $z.p$ é preto.
Agora, mostramos que os casos 2 e 3 mantêm o invariante de laço. (Como acabamos de demonstrar, $z.p$ será preto no próximo teste na linha 1 e o corpo do laço não será executado novamente.)

a. O caso 2 faz z apontar para $z.p$, que é vermelho. Nenhuma mudança adicional em z ou em sua cor ocorre nos casos 2 e 3.
b. O caso 3 torna $z.p$ preto, de modo que, se $z.p$ é a raiz no início da próxima iteração, ele é preto.
c. Como ocorre no caso de 1, as propriedades 1, 3 e 5 são mantidas nos casos 2 e 3.
Visto que o nó z não é a raiz nos casos 2 e 3, sabemos que não há nenhuma violação da propriedade 2. Os casos 2 e 3 não introduzem uma violação da propriedade 2, já que o único nó que se tornou vermelho torna-se um filho de um nó preto pela rotação no caso 3.
Os casos 2 e 3 corrigem a única violação da propriedade 4 e não introduzem outra violação. Mostrando que cada iteração do laço mantém o invariante, também mostramos que RB--INSERT-FIXUP restaura corretamente as propriedades vermelho-preto.

Análise

Qual é o tempo de execução de RB-INSERT? Visto que a altura de uma árvore vermelho-preto em n nós é $O(\lg n)$, as linhas 1−16 de RB-INSERT levam o tempo $O(\lg n)$. Em RB-INSERT-FIXUP, o laço **while** só é repetido se o caso 1 ocorrer, e então o ponteiro z sobe dois níveis na árvore.

Figura 13.6 Casos 2 e 3 do procedimento RB-INSERT-FIXUP. Como no caso 1, a propriedade 4 é violada no caso 2 ou no caso 3 porque z e seu pai $z.p$ são vermelhos. Cada uma das subárvores, α, β, g e d tem uma raiz preta (α, β e g pela propriedade 4, e d porque, caso contrário, estaríamos no caso 1) e cada uma tem a mesma altura preta. Transformamos o caso 2 no caso 3 por uma rotação para a esquerda, o que preserva a propriedade 5: todos os caminhos simples descendentes de um nó até uma folha têm o mesmo número de pretos. O caso 3 provoca algumas mudanças de cores e uma rotação para a direita, o que também preserva a propriedade 5. Em seguida, o laço **while** termina porque a propriedade 4 é satisfeita: não há mais dois nós vermelhos em seguida.

Portanto, o número total de vezes que o laço **while** pode ser executado é $O(\lg n)$. Assim, RB-INSERT demora um tempo total $O(\lg n)$. Além disso, ele nunca executa mais de duas rotações, já que o laço **while** termina se o caso 2 ou o caso 3 for executado.

Exercícios

13.3-1 Na linha 16 de RB-INSERT, atribuímos o nó z recém-inserido com vermelho. Note que, se tivéssemos optado por atribuir z com preto, a propriedade 4 de uma árvore vermelho-preto não seria violada. Por que não optamos por definir z como preto?

13.3-2 Mostre as árvores vermelho-preto que resultam após a inserção sucessiva das chaves 41, 38, 31, 12, 19, 8 em uma árvore vermelho-preto inicialmente vazia.

13.3-3 Suponha que a altura preta de cada uma das subárvores α, β, γ, d, nas Figuras 13.5 e 13.6 seja k. Identifique cada nó em cada figura com sua altura preta para verificar se a transformação indicada preserva a propriedade 5.

13.3-4 O professor Teach está preocupado que RB-INSERT-FIXUP possa atribuir *T.nil.cor* como VERMELHO, caso em que o teste da linha 1 não faria o laço terminar quando z fosse a raiz. Mostre que a preocupação do professor é infundada, demonstrando que RB-INSERT-FIXUP nunca atribui *T.nil.cor* com VERMELHO.

13.3-5 Considere uma árvore vermelho-preto formada pela inserção de n nós com RB-INSERT. Mostre que, se $n > 1$, a árvore tem, no mínimo, um nó vermelho.

13.3-6 Sugira como implementar RB-INSERT de maneira eficiente se a representação para árvores vermelho-preto não incluir nenhum armazenamento para ponteiros superiores.

13.4 ELIMINAÇÃO

Como as outras operações básicas em uma árvore vermelho-preto de n nós, a eliminação de um nó demora o tempo $O(\lg n)$. Eliminar um nó de uma árvore vermelho-preto é um pouco mais complicado que inserir um nó.

O procedimento para eliminar um nó de uma árvore vermelho-preto é baseado no procedimento RB-DELETE (Seção 12.3). Primeiro, precisamos customizar a sub-rotina TRANSPLANT que TREE-DELETE chama, de modo que ela se aplique a uma árvore vermelho-preto:

```
RB-TRANSPLANT(T, u, v)
1   if u.p == T.nil
2       T.raiz = v
3   elseif u == u.p.esquerda
4       u.p.esquerda = v
5   else u.p.direita = v
6   v.p = u.p
```

Há duas diferenças entre o procedimento RB-TRANSPLANT e o procedimento TRANSPLANT. A primeira é que a linha 1 referencia a sentinela *T.nil* em vez de NIL. A segunda é que a atribuição a *.p* na linha 6 ocorre incondicionalmente: podemos atribuir a $v.p$ mesmo que aponte para a sentinela. De fato, exploraremos a capacidade de atribuir a *.p* quando = *T.nil*.

O procedimento RB-DELETE é como o procedimento TREE-DELETE, porém com linhas adicionais de pseudocódigo. Algumas dessas linhas adicionais rastreiam um nó y que poderia causar violações das propriedades vermelho-preto. Quando queremos eliminar o nó z e z tem menos do que dois filhos, z é removido da árvore e queremos que y seja z. Quando z tem dois filhos, y deve ser o sucessor de z , e y passa para a posição de z na árvore. Também lembramos a cor de y antes de ele ser eliminado da árvore ou passar para dentro dela, e rastreamos o nó x que passa para a posição original de y na árvore porque o nó x também poderia causar violações das propriedades vermelho-preto. Após

eliminar o nó z, RB-DELETE chama um procedimento auxiliar RB-DELETE-FIXUP, que muda as cores e executa rotações para restaurar as propriedades vermelho-preto.

```
RB-DELETE(T, z)
1    y = z
2    y-cor-original = y.cor
3    if z.esquerda == T.nil
4          x = z.direita
5          RB-TRANSPLANT(T, z, z.direita)
6    elseif z.direita == T.nil
7          x = z.esquerda
8          RB-TRANSPLANT(T, z, z.esquerda)
9    else y = TREE-MINIMUM(z.direita)
10         y -cor-original = y.cor
11         x = y.direita
12         if y.p == z
13               x.p = y
14         else RB-TRANSPLANT(T, y, y.direita)
15               y.direita = z.direita
16               y.direita.p = y
17         RB-TRANSPLANT(T, z, y)
18         y.esquerda = z.esquerda
19         y.esquerda.p = y
20         y.cor = z.cor
21   if y-cor-original == PRETO
22         RB-DELETE-FIXUP(T, x)
```

Embora RB-DELETE contenha quase duas vezes o número de linhas de pseudocódigo de TREE-DELETE, os dois procedimentos têm a mesma estrutura básica. Podemos encontrar cada linha de TREE-DELETE dentro de RB-DELETE (se substituirmos *T.nil por* NIL e as chamadas a RB-TRANSPLANT por chamadas a TRANSPLANT) se executado sob as mesmas condições.

Apresentamos a seguir, as outras diferenças entre os dois procedimentos:

- Mantemos o nó y como o nó que é retirado da árvore ou que é passado para dentro dela. A linha 1 faz y apontar para o nó z quando z tiver menos que dois filhos e, portanto, é removido. Quando z tem dois filhos, a linha 9 faz y apontar para o sucessor de z exatamente como em TREE-DELETE, e y passa para a posição de z na árvore.
- Como a cor do nó y pode mudar, a variável *y-cor-original* armazena a cor de y antes de ocorrer qualquer mudança. As linhas 2 e 10 definem essa variável imediatamente após atribuições a y. Quando z tem dois filhos, então $y \neq z$ e o nó y passa para a posição original do nó z na árvore vermelho-preto; a linha 20 dá a y a mesma cor de z. Precisamos salvar a cor original de y para testá-la no final de RB-DELETE; se o nó era preto, remover ou mover y poderá causar violações das propriedades vermelho-preto.
- Como discutimos, rastreamos o nó x que passa para a posição original do nó y. As atribuições nas linhas 4, 7 e 11 fazem x apontar para o único filho de y ou, se y não tiver filhos, para a sentinela *T.nil*. (Lembre-se de que dissemos, na Seção 12.3, que y não tem nenhum filho à esquerda.)
- Visto que o nó x passa para a posição original de y, o atributo $x.p$ é sempre definido para apontar para a posição original do pai de y na árvore, mesmo que x seja, de fato, a sentinela *T.nil*. A menos que z seja o pai original de y (o que ocorre somente quando z tiver dois filhos e seu sucessor y for o filho à direita de z), a atribuição a $x.p$ ocorre na linha 6 de RB-TRANSPLANT.

(Observe que, quando RB-TRANSPLANT é chamado nas linhas 5, 8 ou 14, o terceiro parâmetro passado é o mesmo que x.)

Entretanto, quando o pai original de y é z, não queremos que $x.p$ aponte para o pai original de y, visto que estamos eliminando aquele nó da árvore. Como o nó y subirá para ocupar a posição de z na árvore, atribuir y a $x.p$ na linha 13 faz com que $x.p$ aponte para a posição original do pai de y, mesmo que $x = T.nil$.

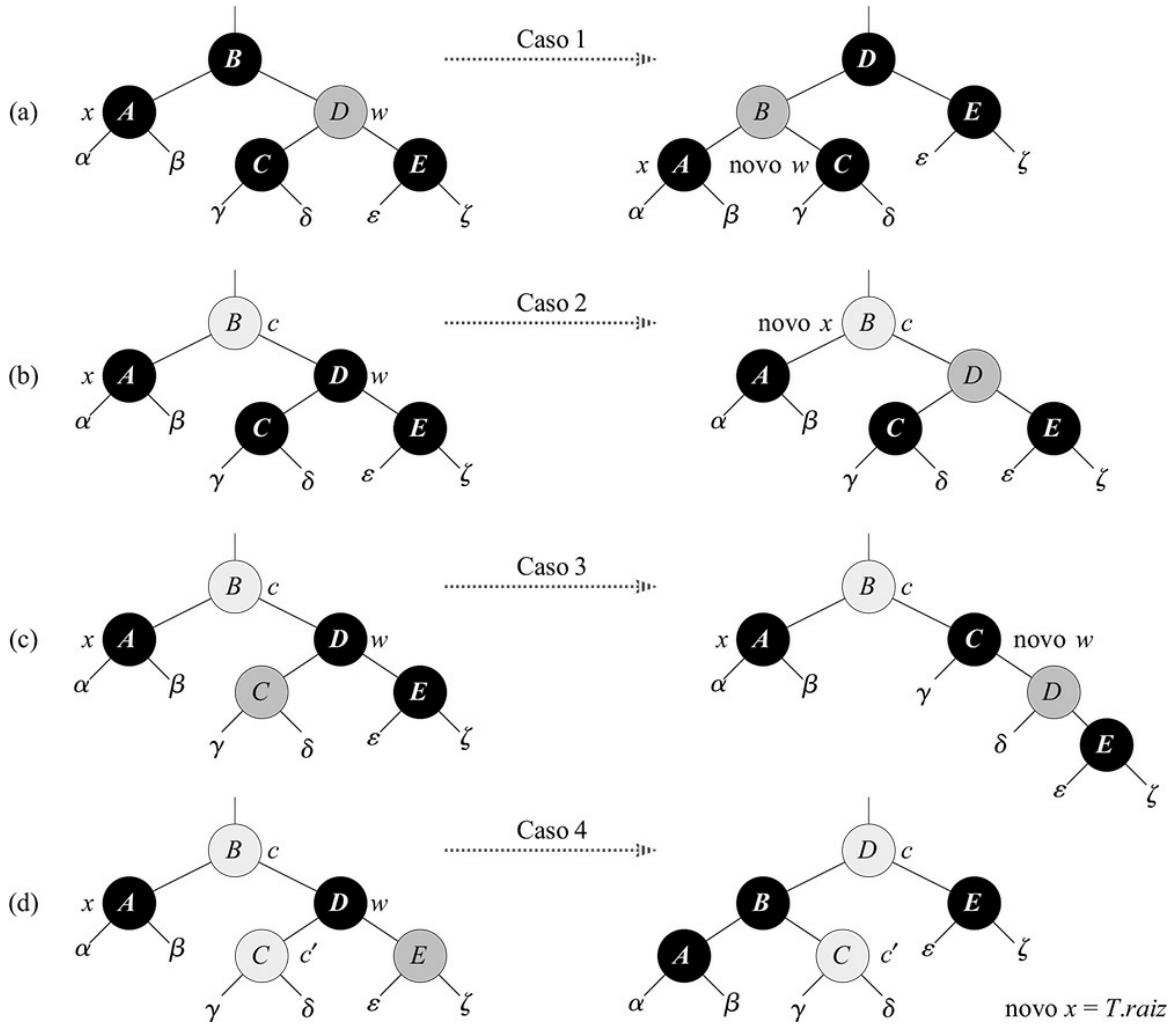
- Por fim, se o nó y era preto, pode ser que tenhamos introduzido uma ou mais violações das propriedades vermelho-preto e, por isso, chamamos RB-DELETE-FIXUP na linha 22 para restaurar as propriedades vermelho-preto. Se y era vermelho, as propriedades vermelho--preto ainda são válidas quando y é eliminado ou movido, pelas seguintes razões:
  1. Nenhuma altura preta na árvore mudou.
  2. Nenhum par de nós vermelhos tornou-se adjacente. Como y toma o lugar de z na árvore, juntamente com a cor de z, não podemos ter dois nós vermelhos adjacentes na nova posição de y na árvore. Além disso, se y não era o filho à direita de z, então x, o filho à direita original de y, substitui y na árvore. Se y é vermelho, então x deve ser preto; portanto, substituir y por x não pode fazer com que dois nós vermelhos se tornem adjacentes.
  3. Visto que y não poderia ter sido a raiz se fosse vermelho, a raiz permanece preta.

Se o nó y era preto, poderão surgir três problemas, que a chamada de RB-DELETE-FIXUP remediará. Primeiro, se y era a raiz e um filho vermelho de y se torna a nova raiz, violamos a propriedade 2. Segundo, se x e $y.p$ (que agora também é $x.p$) eram vermelhos, então violamos a propriedade 4. Terceiro, mover y pela árvore faz com que qualquer caminho simples que continha y anteriormente tenha um nó preto a menos. Assim, a propriedade 5 agora é violada por qualquer ancestral de y na árvore. Podemos corrigir a violação da propriedade 5 dizendo que o nó x, que agora ocupa a posição original de y, tem um preto "extra". Isto é, se somarmos 1 à contagem de nós pretos em qualquer caminho simples que contenha x, então, por essa interpretação, a propriedade 5 se mantém válida. Quando extraímos ou movimentamos o nó preto y, "impomos" sua negritude ao nó x. O problema é que agora o nó x não é nem vermelho nem preto, o que viola a propriedade 1. Em vez disso, o nó x é "duplamente preto" ou "vermelho e preto" e contribui com 2 ou 1, respectivamente, para a contagem de nós pretos em caminhos simples que contêm x. O atributo *cor* de x ainda será VERMELHO (se x é vermelho e preto) ou PRETO (se x é duplamente preto). Em outras palavras, a consequência desse preto extra em um nó é que x apontará para o nó em vez de para o atributo *cor*.

Agora podemos ver o procedimento RB-DELETE-FIXUP e examinar como ele devolve as propriedades vermelho-preto à árvore de busca.

```
RB-DELETE-FIXUP(T; x)
1    while x ≠ T.raiz and x.cor == PRETO
2        if x == x.p.esquerda
3        w = x.p.direita
4        if w.cor == VERMELHO
5            w.cor = PRETO                                   // caso 1
6            x.p.cor = VERMELHO                              // caso 1
7            LEFT-ROTATE(T, x.p)                             // caso 1
8            w = x.p.direita                                 // caso 1
9        if w.esquerda.cor == PRETO and w.direita.cor == PRETO
10           w.cor = VERMELHO                                // caso 2
11           x = x.p                                         // caso 2
12       else if w.direita.cor == PRETO
13           w.esquerda.cor = PRETO                          // caso 3
14           w.cor = VERMELHO                                // caso 3
15           RIGHT-ROTATE(T,w)                               // caso 3
16           w = x.p.direita                                 // caso 3
```

O procedimento RB-DELETE-FIXUP restaura as propriedades 1, 2 e 4. Os Exercícios 13.4-1 e 13.4-2 pedem que você mostre que o procedimento restaura as propriedades 2 e 4 e, assim, no restante desta seção focalizaremos a propriedade 1. O objetivo do laço **while** nas linhas 1−22 é mover o preto extra para cima na árvore até

1. x apontar para um nó vermelho e preto, caso em que colorimos x (isoladamente) de preto na linha 23;
2. x apontar para a raiz, caso em que simplesmente "removemos" o preto extra; ou
3. que, executadas as operações adequadas de rotações e novas colorações, saímos do laço.

Dentro do laço **while**, x sempre aponta para um nó não raiz duplamente preto. Determinamos na linha 2 se x é um filho à esquerda ou um filho à direita de seu pai $x.p$. (Já fornecemos o código para a situação na qual x é um filho à esquerda; a situação na qual x é um filho à direita — linha 22 — é simétrica.) Mantemos um ponteiro w para o irmão de x. Visto que o nó x é duplamente preto, o nó w não pode ser *T.nil* porque, caso contrário, o número de pretos no caminho simples de $x.p$ até a folha w (simplesmente preta) seria menor que o número no caminho simples de $x.p$ até x.

Os quatro casos[2] no código aparecem na Figura 13.7. Antes de examinar cada caso em detalhes, vamos ver, de um modo mais geral, como podemos comprovar que, em cada um dos casos, a transformação preserva a propriedade 5. A ideia-chave é que, em cada caso, a transformação aplicada preserva o número de nós pretos (incluindo o preto extra de x) da raiz da subárvore (inclusive) mostrada até cada uma das subárvores α, β,..., . Assim, se a propriedade 5 é válida antes da transformação, continua a ser válida depois dela. Por exemplo, na Figura 13.7(a), que ilustra o caso 1, o número de nós pretos da raiz até a subárvore α ou β é 3, antes e também depois da transformação. (Mais uma vez, lembre-se de que o nó x adiciona um preto extra.) De modo semelhante, o número de nós pretos da raiz até qualquer das subárvores γ, d, e e é 2, antes e também depois da transformação. Na Figura 13.7(b), a contagem deve envolver o valor c do atributo *cor* da raiz da subárvore mostrada, que pode ser VERMELHO ou PRETO. Se definirmos contador(VERMELHO) = 0 e contador(PRETO) = 1, o número de nós pretos da raiz até α é 2 + contador(c), antes e também depois da transformação. Nesse caso, após a transformação, o novo nó x tem o atributo *cor* c, mas na realidade é vermelho e preto (se c = VERMELHO) ou duplamente preto (se c = PRETO). Os outros casos podem ser verificados de maneira semelhante (veja o Exercício 13.4-5.)

Caso 1: o irmão *w* de *x* é vermelho

O caso 1 (linhas 5−8 de RB-DELETE-FIXUP e Figura 13.7(a)) ocorre quando o nó w, o irmão do nó x, é vermelho. Visto que w deve ter filhos pretos, podemos trocar as cores de w e $x.p$ e depois executar uma rotação para a esquerda em $x.p$ sem violar qualquer das propriedades vermelho-preto. O novo irmão de x, que é um dos filhos de w antes da rotação, agora é preto e, assim, convertemos o caso 1 no caso 2, 3 ou 4.

Os casos 2, 3 e 4 ocorrem quando o nó w é preto; eles são distinguidos pelas cores dos filhos de w.

Figura 13.7 Os casos no laço **while** do procedimento RB-Delete-fixup. Nós em negro têm atributos *cor* preto, nós sombreados em tom mais escuro têm atributos *cor* vermelho e nós sombreados em tom mais claro têm atributos *cor* representados por *c* e *c*', que podem ser vermelho ou preto. As letras α, β, ..., representam subárvores arbitrárias. Cada caso transforma a configuração à esquerda na configuração à direita mudando algumas cores e/ou executando uma rotação. Qualquer nó apontado por *x* tem um preto extra e é duplamente preto ou vermelho e preto. Somente o caso 2 faz o laço se repetir. (**a**) O caso 1 é transformado no caso 2, 3 ou 4 trocando as cores dos nós *B* e *D* e executando uma rotação para a esquerda. (**b**) No caso 2, o preto extra representado pelo ponteiro *x* é deslocado para cima na árvore colorindo o nó *D* de vermelho e ajustando *x* para apontar para o nó *B*. Se entrarmos no caso 2 por meio do caso 1, o laço **while** termina, já que o novo nó *x* é vermelho e preto, e portanto o valor *c* de seu atributo *cor* é VERMELHO. (**c**) O caso 3 é transformado no caso 4 trocando as cores dos nós *C* e *D* e executando uma rotação para a direita. (**d**) O caso 4 remove o preto extra representado por *x* mudando algumas cores e executando uma rotação para a esquerda (sem violar as propriedades vermelho-preto) e, então, o laço termina.

Caso 2: o irmão *w* de *x* é preto e os filhos de *w* são pretos

No caso 2 (linhas 10−11 de RB-Delete-Fixup e Figura 13.7(b)), os filhos de *w* são pretos. Visto que *w* também é preto, retiramos um preto de *x* e também de *w*, deixando *x* com apenas um preto e deixando *w* vermelho. Para compensar a remoção de um preto de *x* e de *w*, gostaríamos de adicionar um preto extra a *x.p*, que era originalmente vermelho ou preto. Fazemos isso repetindo o laço **while** com *x.p* como o novo nó *x*. Observe que, se entrarmos no caso 2 por meio do caso 1, o novo nó *x* será vermelho e preto, já que o *x.p* original era vermelho. Consequentemente, o valor *c* do atributo *cor* do novo nó *x* é vermelho, e o laço termina quando testa a condição de laço. Então colorimos o novo nó *x* de preto (simplesmente) na linha 23.

Caso 3: o irmão *w* de *x* é preto, o filho à esquerda de *w* é vermelho e o filho à direita de *w* é preto

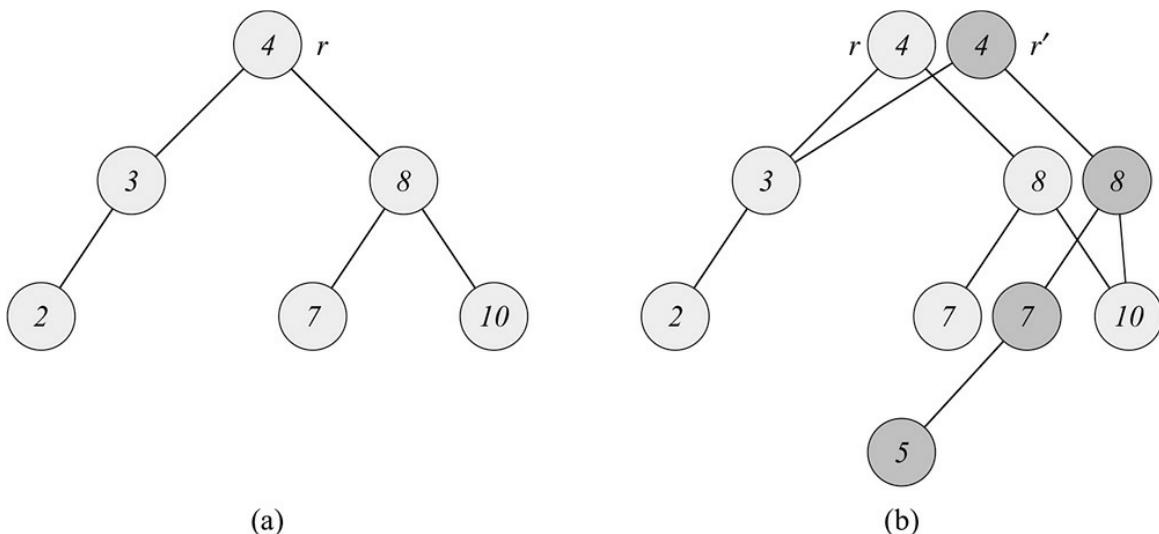
O caso 3 (linhas 13−16 e Figura 13.7(c)) ocorre quando w é preto, seu filho à esquerda é vermelho e seu filho à direita é preto. Podemos permutar as cores de w e de seu filho à esquerda *w.esquerda* e então executar uma rotação para a direita em w sem violar qualquer das propriedades vermelho-preto. O novo irmão w de x é agora um nó preto com um filho à direita vermelho e, assim, transformamos o caso 3 no caso 4.

Caso 4: o irmão *w* de *x* é preto e o filho à direita de *w* é vermelho

O caso 4 (linhas 17−21 e Figura 13.7(d)) ocorre quando o irmão w do nó x é preto e o filho à direita de w é vermelho. Fazendo algumas mudanças de cores e executando uma rotação para a esquerda em $x.p$, podemos remover o preto extra em x, tornando-o unicamente preto, sem violar qualquer das propriedades vermelho-preto. Definir x como a raiz faz o laço **while** terminar ao testar a condição de laço.

Análise

Qual é o tempo de execução de RB-DELETE? Visto que a altura de uma árvore vermelho-preto de n nós é $O(\lg n)$, o custo total do procedimento sem a chamada a RB-DELETE-FIXUP demora o tempo $O(\lg n)$. Dentro de RB-DELETE-FIXUP, cada um dos casos 1, 3 e 4 leva ao término depois de executar um número constante de mudanças de cores e no máximo três rotações. O caso 2 é o único no qual o laço **while** pode ser repetido, e então o ponteiro x se move para cima na árvore no máximo $O(\lg n)$ vezes sem executar nenhuma rotação. Assim, o procedimento RB-DELETE-FIXUP demora o tempo $O(\lg n)$ e executa no máximo três rotações e, portanto, o tempo global para RB-DELETE também é $O(\lg n)$.

Exercícios

13.4-1 Mostre que, após a execução de RB-DELETE-FIXUP, a raiz da árvore tem de ser preta.

13.4-2 Mostre que, se x e $x.p$ são vermelhos em RB-DELETE, então a propriedade 4 é restabelecida pela chamada a RB-DELETE-FIXUP(T, x).

13.4-3 No Exercício 13.3-2, você determinou a árvore vermelho-preto que resulta da inserção sucessiva das chaves 41, 38, 31, 12, 19, 8 em uma árvore inicialmente vazia. Agora, mostre as árvores vermelho-preto que resultam da eliminação sucessiva das chaves na ordem 8, 12, 19, 31, 38, 41.

13.4-4 Em quais linhas do código de RB-DELETE-FIXUP poderíamos examinar ou modificar a sentinela *T.nil?*

13.4-5 Em cada um dos casos da Figura 13.7, dê a contagem de nós pretos da raiz da subárvore mostrada até cada uma das subárvores α, β, ..., , e confirme que cada contagem permanece a mesma depois da transformação. Quando um nó tiver um atributo *cor c* ou *c'*, use a notação contagem(c) ou contagem(c') simbolicamente em sua contagem.

13.4-6 Os professores Skelton e Baron estão preocupados porque, no início do caso 1 de RB-DELETE-FIXUP, o nó $x.p$ poderia não ser preto. Se os professores estão corretos, as linhas 5−6 estão erradas. Mostre que $x.p$ deve ser preto no início do caso 1 e, portanto, os professores não precisam se preocupar.

13.4-7 Suponha que um nó x seja inserido em uma árvore vermelho-preto com RB-INSERT e então imediatamente eliminado com RB-DELETE. A árvore vermelho-preto resultante é igual à árvore vermelho-preto inicial? Justifique sua resposta.

Problemas

13-1 Conjuntos dinâmicos persistentes

Durante o curso de um algoritmo, às vezes, percebemos que precisamos manter versões anteriores de um conjunto dinâmico à medida que ele é atualizado. Tal conjunto é denominado ***persistente***. Um modo de implementar um conjunto persistente é copiar o conjunto inteiro sempre que ele é modificado, mas essa abordagem pode reduzir a velocidade de um programa e também consumir muito espaço. Às vezes, podemos nos sair muito melhor.

Considere um conjunto persistente S com as operações INSERT, DELETE e SEARCH, que implementamos usando árvores de busca binária, como mostra a Figura 13.8(a). Mantemos uma raiz separada para cada versão do conjunto. Para inserir a chave 5 no conjunto, criamos um novo nó com chave 5. Esse nó se torna o filho à esquerda de um novo nó com chave 7, já que não podemos modificar o nó existente com chave 7. De modo semelhante, o novo nó com chave 7 se torna o filho à esquerda de um novo nó com chave 8, cujo filho à direita é o nó existente com chave 10. O novo nó com chave 8 se torna, por sua vez, o filho à direita de uma nova raiz r' com chave 4 cujo filho à esquerda é o nó existente com chave 3. Assim, copiamos apenas parte da árvore e compartilhamos alguns dos nós com a árvore original, como mostra a Figura 13.8(b). Considere que cada nó da árvore tenha os atributos *chave*, *esquerda* e *direita*, mas nenhum pai. (Consulte também o Exercício 13.3-6.)

a. No caso geral de uma árvore de busca binária persistente, identifique os nós que precisamos mudar para inserir uma chave k ou eliminar um nó y.

b. Escreva um procedimento PERSISTENT-TREE-INSERT que, dada uma árvore persistente T e uma chave k a ser inserida, retorne uma nova árvore persistente T' que é o resultado da inserção de k em T.

c. Se a altura da árvore de busca binária persistente T é h, quais são os requisitos de tempo e espaço da sua implementação de PERSISTENT-TREE-INSERT? (O requisito de espaço é proporcional ao número de novos nós alocados.)

d. Suponha que tivéssemos incluído o atributo pai em cada nó. Nesse caso, PERSISTENT-TREE-INSERT precisaria executar cópia adicional. Prove que então, PERSISTENT--TREE-INSERT exigiria tempo e espaço (n), onde n é o número de nós na árvore.

e. Mostre como usar árvores vermelho-preto para garantir que o tempo de execução do pior caso e o espaço são $O(\lg n)$ por inserção ou eliminação.

Figura 13.8 (a) Uma árvore de busca binária com chaves 2, 3, 4, 7, 8, 10. **(b)** A árvore de busca binária persistente que resulta da inserção da chave 5. A versão mais recente do conjunto consiste nos nós acessíveis que partem da raiz r', e a versão anterior consiste nos nós acessíveis a partir de r. Os nós sombreados em tom mais escuro são adicionados quando a chave 5 é inserida.

13-2 Operação de junção em árvores vermelho-preto

A operação de ***junção*** toma dois conjuntos dinâmicos S_1 e S_2 e um elemento x tal que, para qualquer $x_1 \in S_1$ e $x_2 \in S_2$, temos $x_1.chave \leq x.chave \leq x_2.chave$. Ela retorna um conjunto $S = S_1 \cup \{x\} \cup S_2$. Neste problema, investigamos como implementar a operação de junção em árvores vermelho-preto.

a. Dada uma árvore vermelho-preto T, armazenamos sua altura preta como o novo atributo $T.bh$. Mostre que RB-INSERT e RB-DELETE podem manter o atributo bh sem exigir armazenamento extra nos nós da árvore e sem aumentar os tempos de execução assintóticos. Mostre que, enquanto descemos em T, podemos determinar a altura preta de cada nó que visitamos no tempo $O(1)$ por nó visitado.

Desejamos implementar a operação RB-JOIN(T_1, x, T_2), o que destrói T_1 e T_2 e retorna uma árvore vermelho-preto $T = T_1 \cup \{x\} \cup T_2$. Seja n o número de nós em T_1 e T_2.

b. Suponha que $T_1.bh \geq T_2.bh$. Descreva um algoritmo de tempo $O(\lg n)$ que encontre um nó preto y em T_1 com a maior chave entre os nós cuja altura preta é $T_2.bh$.

c. Seja T_y a subárvore com raiz em y. Descreva como $T_y \cup \{x\} \cup T_2$ pode substituir T_y no tempo $O(1)$ sem destruir a propriedade de árvore de busca binária.

d. Que cor x deve ter para que as propriedades vermelho-preto 1, 3 e 5 sejam mantidas? Descreva como impor as propriedades 2 e 4 no tempo $O(\lg n)$.

e. Demonstre que nenhuma generalidade é perdida por adotarmos a premissa na parte (b). Descreva a situação simétrica que surge quando $T_1.bh \leq T_2.bh$.

f. Mostre que o tempo de execução de RB-JOIN é $O(\lg n)$.

13-3 Árvores AVL

Uma ***árvore AVL*** é uma árvore de busca binária de ***altura balanceada***: para cada nó x, a diferença entre as alturas das subárvores à esquerda e à direita de x é no máximo 1. Para implementar uma árvore AVL,

mantemos um atributo extra em cada nó: $x.h$ é a altura do nó x. Como em qualquer outra árvore de busca binária T, supomos que $T.raiz$ aponta para o nó raiz.

a. Prove que uma árvore AVL com n nós tem altura $O(\lg n)$. (*Sugestão:* Prove que uma árvore AVL de altura h tem no mínimo F_h nós, onde F_h é o h-ésimo número de Fibonacci.)

b. No caso da inserção em uma árvore AVL, primeiro colocamos um nó no lugar adequado em ordem de árvore de busca binária. Depois disso, a árvore poderia deixar de ser de altura balanceada. Especificamente, a diferença entre as alturas dos filhos à esquerda e à direita de algum nó poderia ser 2. Descreva um procedimento BALANCE(x), que toma uma subárvore com raiz em x na qual os filhos à esquerda e à direita são de altura balanceada e a diferença entre suas alturas é no máximo 2, isto é, $|h[x.direita] - h[x.esquerda]| \leq 2$, e altera a subárvore com raiz em x de modo que ela se torne de altura balanceada. (*Sugestão:* Use rotações.)

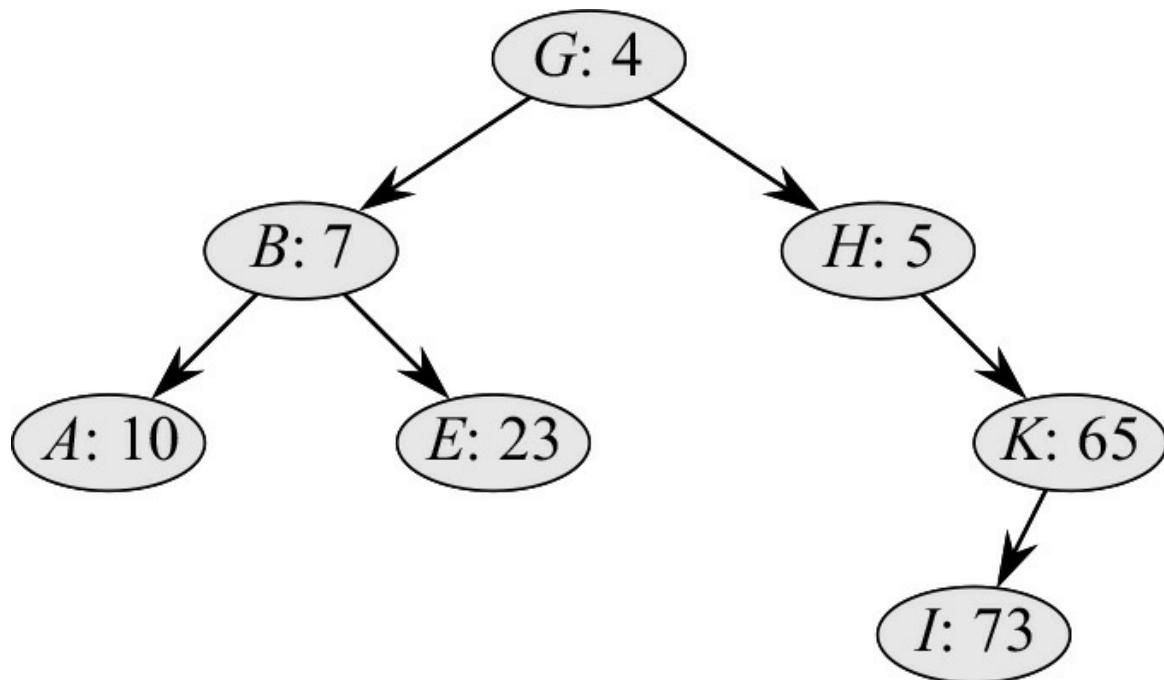
c. Usando a parte (b), descreva um procedimento recursivo AVL-INSERT(x, z), que toma um nó x dentro de uma árvore AVL e um nó z recentemente criado (cuja chave já foi preenchida) e adiciona z à subárvore com raiz em x, mantendo a propriedade de x ser a raiz de uma árvore AVL. Como no procedimento TREE-INSERT da Seção 12.3, considere que $z.chave$ já foi preenchida e que $z.esquerda$ = NIL e $z.direita$ = NIL; considere também que $z.h = 0$. Assim, para inserir o nó z na árvore AVL T, chamamos AVL-INSERT($T.raiz$, z).

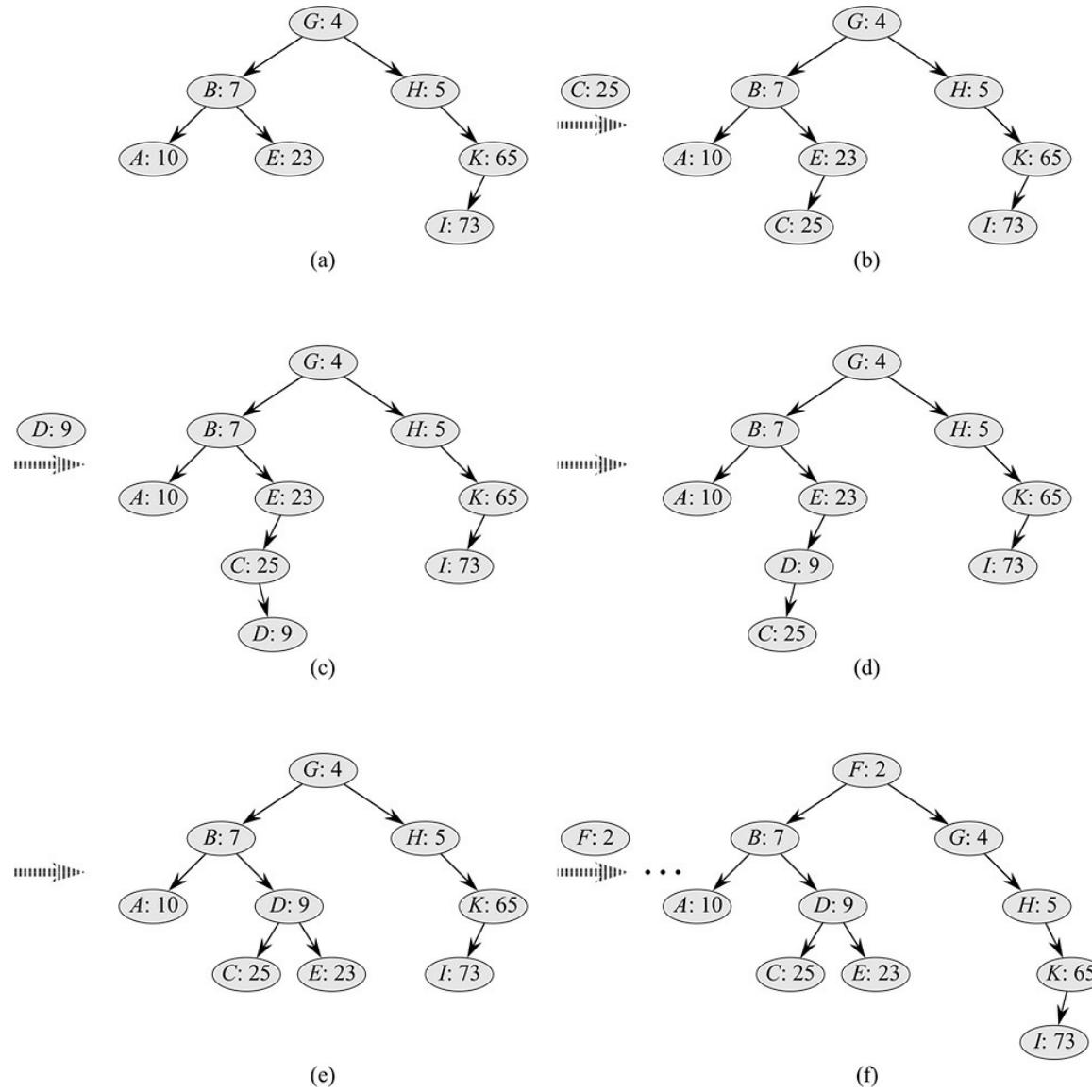
d. Mostre que executar AVL-INSERT em uma árvore AVL de n nós leva o tempo $O(\lg n)$ e efetua $O(1)$ rotações.

13-4 *Treaps*

Se inserirmos um conjunto de n itens em uma árvore de busca binária, a árvore resultante pode ficar terrivelmente desbalanceada, o que resulta em tempos de busca longos. Porém, como vimos na Seção 12.4, árvores de busca binária construídas aleatoriamente tendem a ser balanceadas. Portanto, uma estratégia que, em média, constrói uma árvore balanceada para um conjunto fixo de itens seria permutar aleatoriamente os itens e então inseri-los na árvore nessa ordem.

E se não tivermos todos os itens ao mesmo tempo? Se recebermos os itens um de cada vez, ainda poderemos construir uma árvore de busca binária aleatoriamente com eles? Examinaremos uma estrutura de dados que dá uma resposta afirmativa a essa pergunta. Um ***treap*** é uma árvore de busca binária cujo modo de ordenar os nós é modificado. A Figura 13.9 mostra um exemplo. Como sempre, cada nó x na árvore tem um valor de chave $x.chave$. Além disso, atribuímos $x.prioridade$, que é um número aleatório escolhido independentemente para cada nó. Supomos que todas as prioridades são distintas e também que todas as chaves são distintas. Os nós do treap são ordenados de modo que as chaves obedeçam à propriedade de árvore de busca binária e que as prioridades obedeçam à propriedade de ordem de heap de mínimo:

- Se é um filho à esquerda de u, então $v.chave < u.chave$.
- Se é um filho à direita de u, então $v.chave > u.chave$.
- Se é um filho de u, então $v.prioridade > u.prioridade$.

(Essa combinação de propriedades é o motivo por que a árvore é denominada “treap”; ela tem características de uma árvore — *tree*, em inglês — de busca binária e de um heap.) É conveniente pensar em treaps como descrevemos a seguir. Suponha que inserimos nós $x_1, x_2, ..., x_n$, com chaves associadas, em um treap. Então, o treap resultante é a árvore que teria sido formada se os nós fossem inseridos em uma árvore de busca

binária normal na ordem dada por suas prioridades (escolhidas aleatoriamente), isto é, $x_i.prioridade$ < $x_j.prioridade$ significa que x_i foi inserido antes de x_j.

a. Mostre que, dado um conjunto de nós $x_1, x_2, ..., x_n$, com chaves e prioridades associadas (todas distintas), existe um único treap associado a esses nós.

b. Mostre que a altura esperada de um treap é $Q(\lg n)$ e, consequentemente, o tempo para procurar um valor no treap é $Q(\lg n)$.

Vamos ver como inserir um novo nó em um treap existente. A primeira coisa a fazer é atribuir uma prioridade aleatória ao novo nó. Então, chamamos o algoritmo de inserção, que denominamos TREAP-INSERT, cuja operação está ilustrada na Figura 13.10.

c. Explique como TREAP-INSERT funciona. Explique a ideia em linguagem comum e dê o pseudocódigo. (*Sugestão:* Execute o procedimento habitual de inserção em árvore de busca binária e depois execute rotações para restaurar a propriedade de ordem de heap de mínimo.)

Figura 13.9 Um treap. Cada nó x é identificado com $x.chave$: $x.prioridade$. Por exemplo, a raiz tem chave G e prioridade 4.

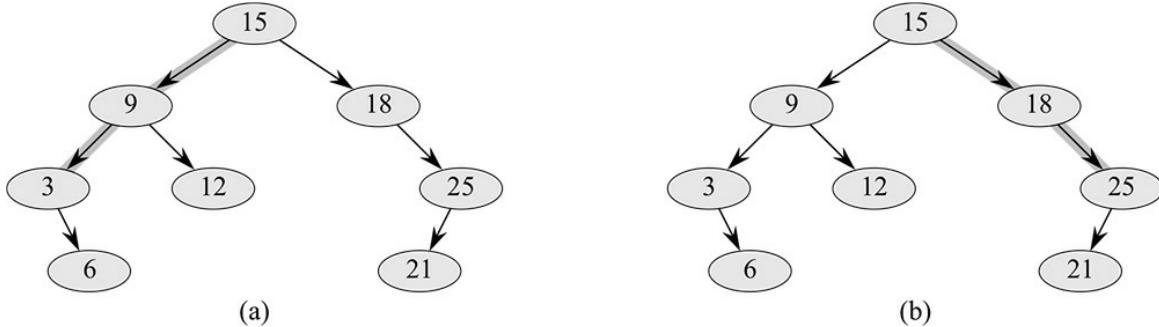



Figura 13.10 A operação de tREAP-INSERT. **(a)** O treap original, antes da inserção. **(b)** O treap depois da inserção de um nó com chave C e prioridade 25. **(c)–(d)** Fases intermediárias quando é inserido um nó com chave D e prioridade 9. **(e)** O treap depois de terminada a inserção das partes (c) e (d). **(f)** O treap depois da inserção de um nó com chave F e prioridade 2.

d. Mostre que o tempo de execução esperado de TREAP-INSERT é $Q(\lg n)$.

TREAP-INSERT executa uma busca e depois uma sequência de rotações. Embora tenham o mesmo tempo de execução esperado, essas duas operações têm custos diferentes na prática. Uma busca lê informações do treap sem modificá-la. Ao contrário, uma rotação muda ponteiros pai e filho dentro do treap. Na maioria dos computadores, operações de leitura são muito mais rápidas que operações de escrita. Assim, seria bom que TREAP-INSERT executasse poucas rotações. Mostraremos que o número esperado de rotações executadas é limitado por uma constante.

Para tal, precisaremos de algumas definições, que estão ilustradas na Figura 13.11. A ***espinha esquerda*** de uma árvore de busca binária T é o caminho simples da raiz até o nó que tenha a menor chave. Em outras palavras, a espinha esquerda é o caminho simples que parte da raiz e consiste apenas em arestas à esquerda. Simetricamente, a ***espinha direita*** de T é o caminho simples que parte da raiz e consiste somente em arestas à direita. O ***comprimento*** de uma espinha é o número de nós que ela contém.

Figura 13.11 Espinhas de uma árvore de busca binária. A espinha esquerda está sombreada em **(a)** e a espinha direita está sombreada em **(b)**.

e. Considere que o treap T imediatamente após TREAP-INSERT inseriu o nó x. Seja C o comprimento da espinha direita da subárvore esquerda de x. Seja D o comprimento da espinha esquerda da subárvore direita de x. Prove que o número total de rotações que foram executadas durante a inserção de x é igual a $C + D$.

Agora calcularemos os valores esperados de C e D. Sem prejuízo da generalidade, consideramos que as chaves são 1, 2, ..., n, já que estamos comparando essas chaves apenas entre si.

Para os nós x e y no treap T, onde $y \neq x$, seja $k = x.chave$ e $i = y.chave$. Definimos variáveis aleatórias indicadoras

$X_{ik} = \mathrm{I}\,\{y$ está na espinha direita da subárvore esquerda de $x\}$.

f. Mostre que $X_{ik} = 1$ se e somente se $y.prioridade > x.prioridade$, $y.chave < x.chave$ e, para todo z tal que $y.chave < z.chave < x.chave$, temos $y.prioridade < z.prioridade$.

g. Mostre que

$$\Pr\{X_{ik} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

h. Mostre que

$$\mathrm{E}[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}.$$

i. Use um argumento de simetria para mostrar que

$$E[D] = 1 - \frac{1}{n-k+1}.$$

j. Conclua que o número esperado de rotações executadas quando um nó é inserido em um treap é menor que 2.

NOTAS DO CAPÍTULO

A ideia de balancear uma árvore de busca se deve a Adelson-Velinski e Landis [2], que apresentaram em 1962 uma classe de árvores de busca balanceadas denominada "árvores AVL", descrita no Problema 13-3. Uma outra classe de árvores de busca, denominada "árvores 2-3", foi apresentada por J. E. Hopcroft (mas não publicada) em 1970. Uma árvore 2-3 mantém o equilíbrio manipulando os graus de nós na árvore. O Capítulo 18 apresenta uma generalização de árvores 2-3 apresentada por Bayer e McCreight [32], denominadas "árvores B".

Árvores vermelho-preto foram criadas por Bayer [34] sob o nome "árvores B binárias simétricas". Guibas e Sedgewick [155] estudaram extensamente suas propriedades e introduziram a convenção de cores vermelho/preto. Andersson [15] dá uma variante de árvores vermelho-preto mais simples de codificar. Weiss [351] chama essa variante de árvores AA. Uma árvore AA é semelhante a uma árvore vermelho-preto, exceto que os filhos à esquerda nunca podem ser vermelhos.

Treaps, o assunto do Problema 13-4, foram propostas por Seidel e Aragon [309]. São a implementação-padrão de um dicionário em LEDA (Library of Efficient Data types and Algorithms) [253], que é uma coleção bem implementada de estruturas de dados e algoritmos.

Há muitas outras variantes em árvores binárias balanceadas, incluindo árvores de peso balanceado [264], árvores de k vizinhos [245] e as árvores de bode expiatório [127]. Talvez as mais curiosas sejam as "árvores oblíquas" ("splay trees") introduzidas por Sleator e Tarjan [320], que são "autoajustáveis". (Uma boa descrição de árvores oblíquas é dada por Tarjan [330].) Árvores oblíquas mantêm equilíbrio sem qualquer condição explícita de equilíbrio, como cor. Em vez disso, "operações oblíquas" (que envolvem rotações) são executadas dentro da árvore toda vez que um acesso é executado. O custo amortizado (veja o Capítulo 17) de cada operação em uma árvore de n nós é $O(\lg n)$.

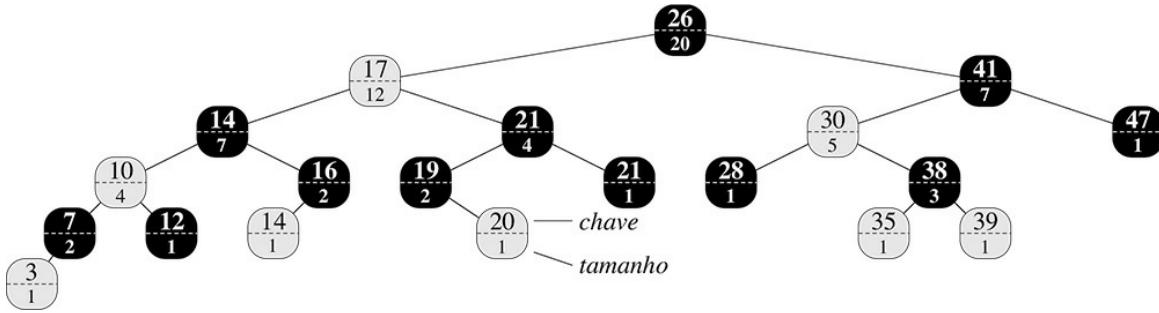
As listas de saltos [286] dão uma alternativa às árvores binárias balanceadas. Uma lista de saltos é uma lista ligada que é ampliada com uma quantidade de ponteiros adicionais. Cada operação de dicionário é executada no tempo esperado $O(\lg n)$ em uma lista de saltos de n itens.

---

1 O caso 2 volta a cair no caso 3 e, portanto, esses dois casos não são mutuamente exclusivos.

2 Como em RB-Insert-Fixup, os casos em RB-Delete-Fixup não são mutuamente exclusivos.

14 AUMENTANDO ESTRUTURAS DE DADOS

Algumas situações de engenharia não exigem mais que uma estrutura de dados "de livro didático" — como uma lista duplamente ligada, uma tabela hash ou uma árvore de busca binária —, mas muitas outras exigem uma pitada de criatividade. Entretanto, apenas em raras situações você precisará criar um tipo inteiramente novo de estrutura de dados. No mais das vezes, será suficiente aumentar uma estrutura de dados comum e nela armazenar informações adicionais. Então, você poderá programar novas operações para que a estrutura de dados suporte a aplicação desejada. Porém, aumentar uma estrutura de dados nem sempre é uma operação direta, já que as informações adicionadas devem ser atualizadas e mantidas pelas operações originais da estrutura.

Este capítulo discute duas estruturas de dados que construímos aumentando as árvores vermelho-preto. A Seção 14.1 descreve uma estrutura de dados que suporta operações gerais de estatísticas de ordem em um conjunto dinâmico. Então, poderemos encontrar rapidamente o i-ésimo menor número em um conjunto ou o posto de um dado elemento na ordenação total do conjunto. A Seção 14.2 abstrai o processo de aumentar uma estrutura de dados e fornece um teorema que pode simplificar o processo de aumento de árvores vermelho-preto. A Seção 14.3 utiliza esse teorema para ajudar a projetar uma estrutura de dados para manter um conjunto dinâmico de intervalos, como intervalos de tempo. Dado um intervalo de consultas, poderemos encontrar rapidamente um intervalo no conjunto que se sobreponha a ele.

14.1 ESTATÍSTICAS DE ORDEM DINÂMICAS

O Capítulo 9 apresentou a noção de estatística de ordem. Especificamente, a i-ésima estatística de ordem de um conjunto de n elementos, onde $i \in \{1, 2, ..., n\}$, é simplesmente o elemento no conjunto que tenha a i-ésima menor chave. Vimos como determinar qualquer estética de ordem no tempo $O(n)$ em um conjunto não ordenado. Nesta seção, veremos como modificar as árvores vermelho-preto para podermos determinar qualquer estatísttica de ordem no tempo $O(\lg n)$. Veremos também como calcular o ***posto*** de um elemento — sua posição na ordem linear do conjunto — no tempo $O(\lg n)$.

A Figura 14.1 mostra uma estrutura de dados que pode suportar operações rápidas de estatísticas de ordem. Uma ***árvore de estatísticas de ordem*** T é simplesmente uma árvore vermelho-preto com informações adicionais armazenadas em cada nó. Além dos atributos habituais da árvore vermelho-preto, $x.chave$, $x.cor$, $x.p$, $x.esquerda$ e $x.direita$ em um nó x, temos outro atributo, $x.tamanho$. Esse atributo contém o número de nós (internos) na subárvore com raiz em x (incluindo o próprio x), isto é, o tamanho da subárvore. Se definirmos o tamanho da sentinela como 0, isto é, se definirmos $tamanho[nil[T$ como 0, então teremos a identidade

$$x.tamanho = x.esquerda.tamanho + x.direita.tamanho + 1 .$$

Figura 14.1 Uma árvore de estatísticas de ordem, que é uma árvore vermelho-preto aumentada. Os nós em cinzento são vermelhos, e os nós em negro são pretos. Além de seus atributos habituais, cada nó x tem um atributo $x.tamanho$, que é o número de nós na subárvore com raiz em x.

Não exigimos que as chaves sejam distintas em uma árvore de estatísticas de ordem. (Por exemplo, a árvore da Figura 14.1 tem duas chaves com valor 14 e duas chaves com valor 21.) Na presença de chaves iguais, a noção de posto que citamos não é bem definida. Eliminamos essa ambiguidade para uma árvore de estatísticas de ordem definindo o posto de um elemento como a posição na qual ele seria impresso em um percurso em in-ordem da árvore. Por exemplo, na Figura 14.1, a chave 14 armazenada em um nó preto tem ordem 5, e a chave 14 armazenada em um nó vermelho tem ordem 6.

Recuperação de um elemento com determinado posto

Antes de mostrar como manter as informações de tamanho durante inserção e eliminação, vamos examinar a implementação de duas consultas de estatísticas de ordem que utilizam essas informações adicionais. Começamos com uma operação que recupera um elemento com um determinado posto. O procedimento OS-SELECT(x, i) retorna um ponteiro para o nó que contém a i-ésima menor chave na subárvore com raiz em x. Para encontrar o nó com a i-ésima menor chave em uma árvore de estatísticas de ordem T, chamamos OS-SELECT($raiz[T]$, i).

```
OS-SELECT(x, i)
1  r = x.esquerda.tamanho + 1
2  if i = r
3     return x
4  elseif i < r
5     return OS-SELECT(x.esquerda, i)
6  else return OS-SELECT(x.direita, i − r)
```

Na linha 1 de OS-SELECT, calculamos r, o posto do nó x dentro da subárvore com raiz em x. O valor de $x.esquerda.tamanho$ é o número de nós que vêm antes de x em um percurso de árvore, em um percurso em in-ordem da subárvore com raiz em x. Assim, $x.esquerda.tamanho + 1$ é o posto de x dentro da subárvore com raiz em x. Se $i = r$, então o nó x é o i-ésimo menor elemento, e assim retornamos x na linha 3. Se $i < r$, então o i-ésimo menor elemento encontra-se na subárvore esquerda de x e, portanto, fazemos recursão em $x.esquerda$ na linha 5. Se $i > r$, então o i-ésimo menor elemento encontra-se na subárvore direita de x. Visto que a subárvore com raiz em x contém r elementos que vêm antes da subárvore direita de x em um percurso de árvore em ordem, o i-ésimo menor elemento na subárvore com raiz em x é o $(i - r)$-ésimo menor elemento na subárvore com raiz em $x.direita$. A linha 6 determina esse elemento recursivamente.

Para ver como OS-SELECT funciona, considere uma busca pelo 17º menor elemento na árvore de estatísticas de ordem da Figura 14.1. Começamos com x como a raiz, cuja chave é 26, e com $i = 17$. Como o tamanho da subárvore esquerda de 26 é 12, seu posto é 13. Assim, sabemos que o nó com posto 17 é o $17 - 13 = 4$º menor elemento na subárvore direita de 26. Após a chamada recursiva, x é o nó com chave 41 e $i = 4$. Visto que o tamanho da subárvore esquerda de 41 é 5, seu posto dentro da sua subárvore é 6. Portanto, sabemos que o nó com posto 4 está no 4º menor

elemento da subárvore esquerda de 41. Após a chamada recursiva, x é o nó com chave 30 e seu posto dentro de sua subárvore é 2. Assim, faremos mais uma vez uma recursão para encontrar o $4 - 2 = 2$º menor elemento na subárvore com raiz no nó com chave 38. Agora, descobrimos que sua subárvore esquerda tem tamanho 1, o que significa que ele é o segundo menor elemento. Assim, o procedimento retorna um ponteiro para o nó com chave 38.

Como cada chamada recursiva desce um nível na árvore de estatísticas de ordem, o tempo total para OS-SELECT é, na pior das hipóteses, proporcional à altura da árvore. Visto que se trata de uma árvore vermelho-preto, sua altura é $O(\lg n)$, onde n é o número de nós. Assim, o tempo de execução de OS-SELECT é $O(\lg n)$ para um conjunto dinâmico de n elementos.

Determinação do posto de um elemento

Dado um ponteiro para um nó x em uma árvore de estatísticas de ordem T, o procedimento OS-RANK retorna a posição de x na ordem linear determinada por um percurso em in-ordem da árvore T.

```
OS-RANK(T, x)
1  r = x.esquerda.tamanho + 1
2  y = x
3  while y ≠ T.raiz
4      if y = y.p.direita
5              r = r + y.p.esquerda.tamanho + 1
6      y = y.p
7  return r
```

O procedimento funciona da maneira descrita a seguir. Podemos considerar o posto de x como o número de nós que precedem x em um percurso de árvore em in-ordem, mais 1 para o próprio x. OS-RANK mantém o seguinte invariante de laço:

No início de cada iteração do laço **while** das linhas 3 a 6, r é o posto de $x.chave$ na subárvore com raiz no nó y.

Usamos esse invariante de laço para mostrar que OS-RANK funciona corretamente como a seguir:

Inicialização: Antes da primeira iteração, a linha 1 atribui a r o posto de $x.chave$ dentro da subárvore com raiz em x. Fazer $y = x$ na linha 2 torna o invariante verdadeiro na primeira vez que o teste na linha 3 é executado.

Manutenção: No fim de cada iteração do laço **while**, atribuímos $y = y.p$. Assim, devemos mostrar que, se r é o posto de $x.chave$ na subárvore com raiz em y no início do corpo do laço, então r é o posto de $x.chave$ na subárvore com raiz em $y.p$ no fim do corpo do laço. Em cada iteração do laço **while**, consideramos a subárvore com raiz em $y.p$. Já contamos o número de nós na subárvore com raiz no nó y que precedem x em um percurso em in-ordem; assim, devemos adicionar os nós na subárvore com raiz no irmão de y que precedem x em um percurso em in-ordem, mais 1 para $y.p$, se ele também precede x. Se y é um filho da esquerda, nem $y.p$ nem qualquer nó na subárvore direita de $y.p$ precede x; portanto, deixamos r como está. Caso contrário, y é um filho à direita e todos os nós na subárvore esquerda de $y.p$ precedem x, assim como o próprio$y.p$. Portanto, na linha 5, adicionamos $y.p.esquerda.tamanho + 1$ ao valor atual de r.

Término: O laço termina quando $y = T.raiz$, de modo que a subárvore com raiz em y é a árvore inteira. Assim, o valor de r é o posto de $x.chave$ na árvore toda.

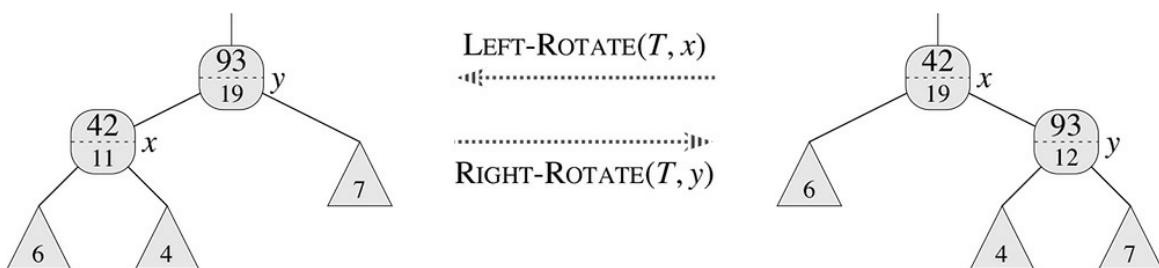
Como exemplo, quando executamos OS-RANK na árvore de estatísticas de ordem da Figura 14.1 para encontrar o posto do nó com chave 38, obtemos a seguinte sequência de valores de $y.chave$ e r no início do laço **while**:

iteração	*y.chave*	*r*
1	38	2
2	30	4
3	41	4
4	26	17

O procedimento retorna o posto 17.

Visto que cada iteração do laço **while** leva o tempo $O(1)$ e y sobe um nível na árvore com cada iteração, o tempo de execução de OS-RANK é, na pior das hipóteses, proporcional à altura da árvore: $O(\lg n)$ em uma árvore de estatísticas de ordem de n nós.

Manutenção de tamanhos de subárvores

Dado o atributo *tamanho* em cada nó, OS-SELECT e OS-RANK podem calcular rapidamente informações de estatísticas de ordem. Porém, a menos que possamos manter eficientemente esses atributos dentro das operações modificadoras básicas em árvores vermelho-preto, nosso trabalho terá sido em vão. Agora, mostraremos como manter tamanhos de subárvores para inserção e eliminação sem afetar os tempos de execução assintóticos de qualquer das operações.

Observamos, na Seção 13.3, que a inserção em uma árvore vermelho-preto consiste em duas fases. A primeira fase percorre a árvore de cima para baixo a partir da raiz, inserindo o novo nó como um filho de um nó existente. A segunda fase sobe a árvore, alterando cores e executando rotações para manter as propriedades vermelho-preto.

Para manter os tamanhos das subárvores na primeira fase, simplesmente incrementamos *x.tamanho* para cada nó x no caminho descendente simples percorrido da raiz até às folhas. O novo nó adicionado obtém um *tamanho* igual a 1. Visto que existem $O(\lg n)$ nós no caminho percorrido, o custo adicional de manter os atributos *tamanho* é $O(\lg n)$.

Na segunda fase, as únicas mudanças estruturais na árvore vermelho-preto subjacente são causadas por rotações, das quais existem no máximo duas. Além disso, uma rotação é uma operação local: somente dois nós têm seus atributos *tamanho* invalidados. A ligação em torno da qual a rotação é executada incide nesses dois nós. Referindo-nos ao código de LEFT-ROTATE (T, x) na Seção 13.2, adicionamos as seguintes linhas:

12*y.tamanho* = *x.tamanho*
13*x.tamanho* = *x.esquerda.tamanho* + *x.direita.tamanho* + 1

A Figura 14.2 ilustra como os atributos são atualizados. A mudança em RIGHT-ROTATE é simétrica.

Figura 14.2 Atualização de tamanhos de subárvores durante rotações. A ligação em torno da qual a rotação é executada é incidente nos dois nós cujos atributos *tamanho* precisam ser atualizados. As atualizações são locais, exigindo apenas as informações *tamanho* armazenadas em x, y e nas raízes das subárvores mostradas como triângulos.

Visto que são executadas no máximo duas rotações durante a inserção em uma árvore vermelho-preto, gastamos somente o tempo adicional $O(1)$ na atualização de atributos *tamanho* na segunda fase. Portanto, o tempo total para inserção em uma árvore de estatísticas de ordem de n nós é $O(\lg n)$ — assintoticamente igual ao de uma árvore vermelho-preto comum.

A eliminação em uma árvore vermelho-preto também consiste em duas fases: a primeira age na árvore de busca subjacente, e a segunda provoca no máximo três rotações; fora isso, não executa nenhuma mudança estrutural (consulte a Seção 13.4). A primeira fase extrai um nó y da árvore ou move esse nó para cima dentro da árvore. Para atualizar os tamanhos das subárvores, simplesmente percorremos um caminho simples do nó y (começando em sua posição original dentro da árvore) até a raiz, decrementando o atributo *tamanho* de cada nó no caminho. Visto que esse caminho tem o comprimento $O(\lg n)$ em uma árvore vermelho-preto de n nós, o tempo adicional despendido na manutenção de atributos *tamanho* na primeira fase é $O(\lg n)$. Tratamos as $O(1)$ rotações na segunda fase de eliminação da mesma maneira que a inserção. Assim, tanto a inserção quanto a eliminação, incluindo a manutenção dos atributos *tamanho*, demoram o tempo $O(\lg n)$ para uma árvore de estatísticas de ordem de n nós.

Exercícios

14.1-1 Mostre como OS-SELECT($T.raiz$, 10) funciona na árvore vermelho-preto T da Figura 14.1.

14.1-2 Mostre como OS-RANK(T, x) funciona na árvore vermelho-preto T da Figura 14.1 e no nó x com $x.chave$ = 35.

14.1-3 Escreva uma versão não recursiva de OS-SELECT.

14.1-4 Escreva um procedimento recursivo OS-KEY-RANK(T, k) que tome como entrada uma árvore de estatísticas de ordem T e uma chave k e retorne o posto de k no conjunto dinâmico representado por T. Suponha que as chaves de T sejam distintas.

14.1-5 Dado um elemento x em uma árvore de estatísticas de ordem de n nós e um número natural i, de que modo podemos determinar o i-ésimo sucessor de x na ordem linear da árvore no tempo $O(\lg n)$?

14.1-6 Observe que, sempre que referenciamos o atributo *tamanho* de um nó em OS-SELECT ou OS-RANK, o usamos somente para calcular um posto. De acordo com isso, suponha que armazenemos em cada nó seu posto na subárvore da qual ele é a raiz. Mostre como manter essas informações durante inserção e eliminação. (Lembre-se de que essas duas operações podem provocar rotações.)

14.1-7 Mostre como usar uma árvore de estatísticas de ordem para contar o número de inversões (ver o Problema 2-4) em um arranjo de tamanho n no tempo $O(n \lg n)$.

14.1-8 ★ Considere n cordas em um círculo, cada uma definida por suas extremidades. Descreva um algoritmo de tempo $O(n \lg n)$ para determinar o número de pares de cordas que se interceptam no interior do círculo. (Por exemplo, se as n cordas são todas diâmetros que se encontram no centro, então a resposta correta é $\binom{n}{2}$.) Suponha que nenhum par de cordas compartilhe um ponto extremo.

14.2 COMO AUMENTAR UMA ESTRUTURA DE DADOS

O processo de aumentar uma estrutura de dados básica para suportar funcionalidade adicional ocorre com bastante frequência no projeto de algoritmos. Nós o usaremos novamente na próxima seção para projetar uma estrutura

de dados que suporta operações em intervalos. Nesta seção, examinaremos as etapas envolvidas em tal aumento. Também provaremos um teorema que nos permite aumentar árvores vermelho-preto facilmente em muitos casos.

Podemos dividir o processo de aumento de uma estrutura de dados em quatro etapas:

1. Escolher uma estrutura de dados subjacente.
2. Determinar informações adicionais que devem ser mantidas na estrutura de dados subjacente.
3. Verificar se podemos manter as informações adicionais para as operações modificadoras básicas na estrutura de dados subjacente.
4. Desenvolver novas operações.

Como ocorre com qualquer método de projeto prescritivo, você não deve seguir cegamente as etapas na ordem dada. Praticamente todo trabalho de projeto contém um elemento de tentativa e erro e, de modo geral, todas as etapas são executadas em paralelo. Por exemplo, não tem sentido algum determinar informações adicionais e desenvolver novas operações (etapas 2 e 4) se não conseguirmos manter as informações adicionais eficientemente. Apesar disso, esse método de quatro etapas dá um bom foco para seus esforços de aumentar uma estrutura de dados e é também um bom modo de organizar a documentação de uma estrutura de dados aumentada.

Seguimos essas etapas na Seção 14.1 para projetar nossas árvores de estatísticas de ordem. Na etapa 1, escolhemos as árvores vermelho-preto como a estrutura de dados subjacente. Uma pista para determinar a adequação de árvores vermelho-preto vem de seu suporte eficiente para outras operações de conjuntos dinâmicos em uma ordem total, como MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR.

Na etapa 2, acrescentamos o atributo *tamanho*, no qual cada nó x armazena o tamanho da subárvore com raiz em x. Em geral, as informações adicionais melhoram a eficiência das operações. Por exemplo, poderíamos ter implementado OS-SELECT e OS-RANK usando apenas as chaves armazenadas na árvore, mas eles não teriam sido executados em tempo $O(\lg n)$. Algumas vezes, as informações adicionais são informações de ponteiros e não de dados, como no Exercício 14.2-1. Na etapa 3, asseguramos que a inserção e a eliminação poderiam manter os atributos *tamanho* e ainda assim serem executadas no tempo $O(\lg n)$. No caso ideal, precisaríamos atualizar somente alguns elementos da estrutura de dados para manter as informações adicionais. Por exemplo, se simplesmente armazenássemos em cada nó o posto que ele ocupa na árvore, os procedimentos OS-SELECT e OS-RANK seriam executados rapidamente, mas inserir um novo elemento mínimo causaria uma mudança nessas informações em todos os nós da árvore. Porém, quando armazenamos tamanhos de subárvores, inserir um novo elemento provoca mudanças nas informações de apenas $O(\lg n)$ nós.

Na etapa 4, desenvolvemos as operações OS-SELECT e OS-RANK. Afinal, a necessidade de novas operações é o motivo pelo qual nos preocupamos em aumentar uma estrutura de dados. Ocasionalmente, em vez de desenvolver novas operações, usamos as informações adicionais para acelerar operações existentes, como no Exercício 14.2-1.

Como aumentar árvores vermelho-preto

Quando árvores vermelho-preto formam a base de uma estrutura de dados aumentada, podemos provar que inserção e eliminação sempre podem manter eficientemente certos tipos de informações adicionais, o que torna a etapa 3 muito fácil. A prova do teorema a seguir é semelhante ao argumento da Seção 14.1 de que podemos manter o atributo *tamanho* em árvores de estatísticas de ordem.

Teorema 14.1 (Como aumentar uma árvore vermelho-preto)

Seja f um atributo que aumenta uma árvore vermelho-preto T de n nós, e suponha que o valor de f para cada nó x dependa somente das informações nos nós x, $x.esquerda$ e $x.direita$, incluindo possivelmente $x.esquerda.f$ e $x.direita.f$. Então, podemos manter os valores de f em todos os nós de T durante inserção e eliminação, sem afetar assintoticamente o desempenho $O(\lg n)$ dessas operações.

Prova A principal ideia da prova é que uma mudança em um atributo f em um nó x se propaga apenas até os ancestrais de x na árvore. Isto é, mudar $x.f$ pode exigir que $x.p.f$ seja atualizado, mas nada além disso; atualizar $x.p.f$ pode exigir que $x.p.p.f$ seja atualizado, mas nada além disso, e assim por diante subindo a árvore. Uma vez atualizado $T.raiz.f$, nenhum outro nó dependerá do novo valor e, assim, o processo termina. Visto que a altura de uma árvore vermelho-preto é $O(\lg n)$, mudar um atributo f em um nó custa o tempo $O(\lg n)$ na atualização de todos os nós que dependem da mudança.

A inserção de um nó x em T consiste em duas fases (consulte a Seção 13.3). A primeira fase insere x como um filho de um nó existente $x.p$. Podemos calcular o valor de $x.f$ no tempo $O(1)$ já que, por hipótese, ele depende apenas das informações nos outros atributos do próprio x e das informações nos filhos de x, mas os filhos de x são a sentinela $T.nil$. Uma vez calculado $x.f$, a mudança se propaga para cima na árvore. Assim, o tempo total para a primeira fase de inserção é $O(\lg n)$. Durante a segunda fase, as únicas mudanças estruturais na árvore vêm de rotações. Visto que apenas dois nós mudam em uma rotação, o tempo total para atualizar os atributos f é $O(\lg n)$ por rotação. Como o número de rotações durante a inserção é no máximo dois, o tempo total para inserção é $O(\lg n)$.

Como a inserção, a eliminação tem duas fases (consulte a Seção 13.4). Na primeira fase, as mudanças na árvore ocorrem quando o nó eliminado é removido da árvore. Se o nó eliminado tiver dois filhos naquele momento, seu sucessor passará para a posição do nó eliminado. A propagação até f das atualizações causadas por essas mudanças custa, no máximo, $O(\lg n)$, já que as mudanças na árvore são locais. Corrigir a árvore vermelho-preto durante a segunda fase requer no máximo três rotações, e cada rotação requer no máximo o tempo $O(\lg n)$ para propagar as atualizações até f. Portanto, como a inserção, o tempo total para eliminação é $O(\lg n)$.

Em muitos casos, tal como a manutenção de atributos *tamanho* em árvores de estatísticas de ordem, o custo de atualizar após uma rotação é $O(1)$, em vez do custo $O(\lg n)$ deduzido na prova do Teorema 14.1. O Exercício 14.2-4 dá um exemplo.

Exercícios

14.2-1 Mostre, adicionando ponteiros aos nós, como suportar cada uma das consultas de conjuntos dinâmicos MINIMUM, MAXIMUM, SUCCESSOR e PREDEDECESSOR no tempo de pior caso $O(1)$ em uma árvore de estatísticas de ordem aumentada. O desempenho assintótico de outras operações em árvores de estatísticas de ordem não deve ser afetado.

14.2-2 Podemos manter as alturas pretas de nós em uma árvore vermelho-preto como atributos nos nós da árvore sem afetar o desempenho assintótico de qualquer das operações de árvores vermelho-preto? Mostre como, se sua resposta for positiva, ou justifique uma resposta negativa. E se quiséssemos manter a profundidade dos nós?

14.2-3 ★ Seja $\otimes$ um operador binário associativo e seja a um atributo mantido em cada nó de uma árvore vermelho-preto. Suponha que queiramos incluir em cada nó x um atributo adicional f tal que $f[x] = x_1.a \otimes x_2.a \otimes \ldots \otimes x_m.a$, onde $x_1, x_2, \ldots, x_m$ é a listagem em ordem de nós da subárvore com raiz em x. Mostre como atualizar os atributos f em tempo $O(1)$ após uma rotação. Modifique ligeiramente seu argumento para aplicá-lo aos atributos *tamanho* em árvores de estatísticas de ordem.

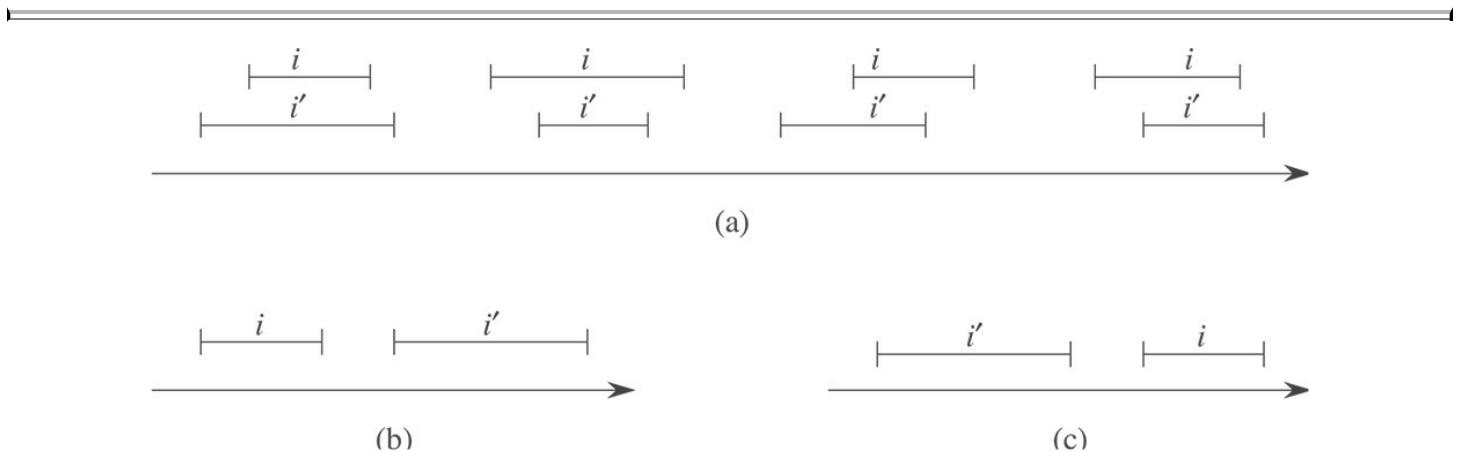
14.2-4 ★ Desejamos aumentar árvores vermelho-preto com uma operação RB-ENUMERATE(x, a, b) que produza todas as chaves k tais que $a \leq k \leq b$ em uma árvore vermelho-preto com raiz em x. Descreva como implementar RB-ENUMERATE em tempo Q$(m + \lg n)$, onde m é o número de chaves na saída e n é o número de nós internos na árvore. (*Sugestão*: Não há necessidade de adicionar novos atributos à árvore vermelho-preto.)

14.3 ÁRVORES DE INTERVALOS

Nesta seção, ampliaremos as árvores vermelho-preto para suportar operações em conjuntos dinâmicos de intervalos. Um ***intervalo fechado*** é um par ordenado de números reais $[t_1, t_2]$, com $t_1, \leq t_2$. O intervalo $[t_1, t_2]$ representa o conjunto $\{t \in \quad : t_1, \leq t \leq t_2\}$. Intervalos ***abertos*** e ***semi-abertos*** excluem ambas ou uma das extremidades do conjunto, respectivamente. Nesta seção, suporemos que os intervalos são fechados; a extensão dos resultados a intervalos abertos e semi-abertos é conceitualmente direta.

Intervalos são convenientes para representar eventos tais que cada um ocupe um período contínuo de tempo. Por exemplo, poderíamos querer consultar um banco de dados de intervalos de tempo para descobrir quais eventos ocorreram durante um determinado intervalo. A estrutura de dados nesta seção fornece um meio eficiente para manter esse banco de dados de intervalos.

Podemos representar um intervalo $[t_1, t_2]$ como um objeto i, com atributos $i.baixo = t_1$ (o ***ponto extremo baixo***) e $i.alto = t_2$ (o ***ponto extremo alto***). Dizemos que os intervalos i e i' se ***sobrepõem*** se $i \cap i' \neq \varnothing$, isto é, se $i.baixo \leq i'.alto$ e $i'.baixo \leq i.alto$. Como mostra a Figura 14.3, quaisquer dois intervalos i e i' satisfazem a ***tricotomia de intervalos***; isto é, exatamente uma das três propriedades a seguir é válida:

a. i e i' se sobrepõem.
b. i está à esquerda de i' (isto é, $i.alto < i'.baixo$).
c. i está à direita de i' (isto é, $i'.alto < i.baixo$).

Uma **árvore de intervalos** é uma árvore vermelho-preto que mantém um conjunto dinâmico de elementos, sendo que cada elemento x contém um intervalo $x.int$. Árvores de intervalos suportam as seguintes operações:

INTERVAL-INSERT(T, x) acrescenta o elemento x à árvore de intervalos T. Supõe-se que o atributo *int* desse elemento contém um intervalo.

INTERVAL-DELETE(T, x) remove o elemento x da árvore de intervalos T.

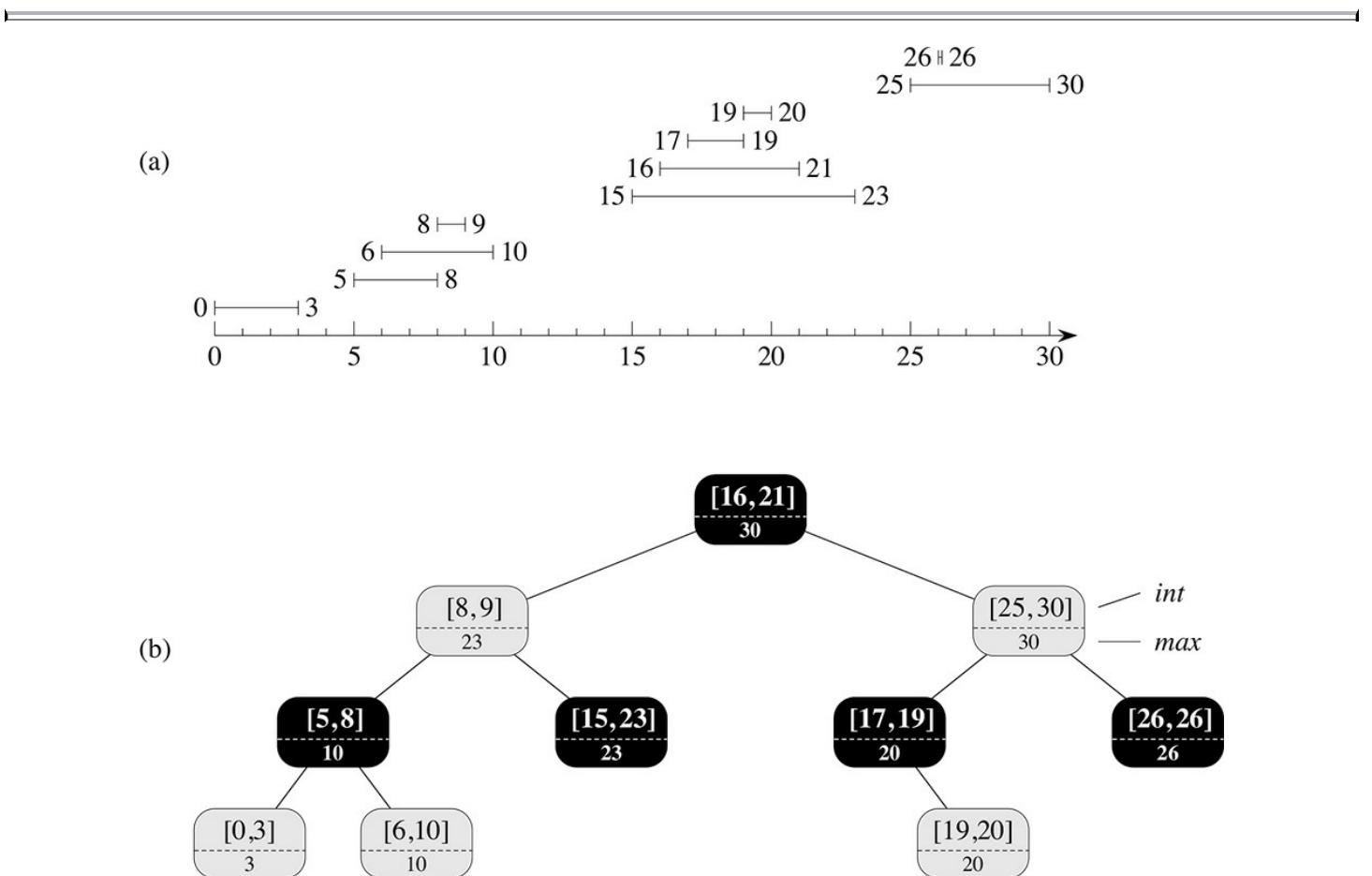
INTERVAL-SEARCH(T, i) retorna um ponteiro para um elemento x na árvore de intervalos T tal que $x.int$ se sobrepõe ao intervalo i ou um ponteiro para a sentinela $T.nil$, se não existir tal elemento no conjunto.

A Figura 14.4 mostra como uma árvore de intervalos representa um conjunto de intervalos. Acompanharemos o método de quatro etapas da Seção 14.2 ao mesmo tempo que revisamos o projeto de uma árvore de intervalos e as operações nela executadas.

Figura 14.3 A tricotomia de intervalos para dois intervalos fechados i e i'. (**a**) Se i e i' se sobrepõem, há quatro situações; em cada uma, $i.baixo \leq i'.alto$ e $i'.baixo \leq i.alto$. (**b**) Os intervalos não se sobrepõem, e $i.alto < i'.baixo$. (**c**) Os intervalos não se sobrepõem, e $i'.alto < i'.baixo$.

Etapa 1: Estrutura de dados subjacente

Escolhemos uma árvore vermelho-preto na qual cada nó x contém um intervalo $x.int$ e a chave de x é o ponto extremo baixo, $x.int.baixo$, do intervalo. Assim, um percurso de árvore em in-ordem pela estrutura de dados produz uma lista de intervalos em sequência ordenada pelo ponto extremo menor.

Etapa 2: Informações adicionais

Além dos próprios intervalos, cada nó x contém um valor $x.max$, que é o valor máximo de qualquer ponto extremo de intervalo armazenado na subárvore com raiz em x.

Etapa 3: Manutenção das informações

Temos de verificar que inserção e eliminação em uma árvore de intervalos de n nós demoram o tempo $O(\lg n)$. Podemos determinar $x.max$ dado o intervalo $x.int$ e os valores max dos filhos do nó x:

$$x.max = \max(x.int.alto, x.esquerda.max, x.direita.max) .$$

Assim, pelo Teorema 14.1, inserção e eliminação são executadas no tempo $O(\lg n)$. De fato, podemos atualizar os atributos max após uma rotação no tempo $O(1)$, como mostram os Exercícios 14.2-3 e 14.3-1.

Figura 14.4 Uma árvore de intervalos. (**a**) Um conjunto de 10 intervalos mostrados em sequência ordenada de baixo para cima por ponto extremo esquerdo. (**b**) A árvore de intervalos que os representa. Cada nó x contém um intervalo, mostrado acima da linha tracejada, e o máximo valor de qualquer ponto extremo de intervalo na subárvore com raiz em x, mostrado abaixo da linha tracejada. Um percurso em in-ordem da árvore produz uma lista de nós em sequência ordenada por ponto extremo esquerdo.

Etapa 4: Desenvolvimento de novas operações

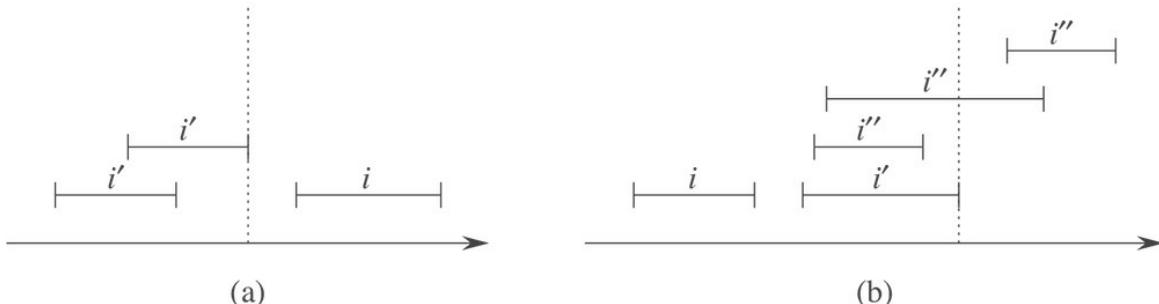
A única operação nova de que necessitamos é INTERVAL-SEARCH(T, i), que encontra um nó na árvore T cujo intervalo se sobrepõe ao intervalo i. Se não existir nenhum intervalo que se sobreponha a i na árvore, o procedimento retorna um ponteiro para a sentinela $nil[T]$.

```
INTERVAL-SEARCH(T, i)

1   x = T.raiz
2   while x ≠ T.nil e i não se sobrepõe a x.int
3      if x.esquerda ≠ T.nil e x.esquerda.max ≥ i.baixo
4              x = x.esquerda
5      else x = x.direita
6   return x
```

A busca de um intervalo que se sobreponha a i começa com x na raiz da árvore, prossegue no sentido descendente e termina quando encontra um intervalo sobreposto ou quando x aponta para a sentinela $T.nil$. Como cada iteração do laço básico demora o tempo $O(1)$ e visto que a altura de uma árvore vermelho-preto de n nós é $O(\lg n)$, o procedimento INTERVAL-SEARCH demora o tempo $O(\lg n)$.

Antes de vermos por que INTERVAL-SEARCH é correto, vamos examinar como ele funciona na árvore de intervalos da Figura 14.4. Vamos supor que desejamos encontrar um intervalo que se sobreponha ao intervalo i = [22, 25]. Começamos com o nó x como raiz, o qual contém [16, 21] e não se sobrepõe a i. Visto que $x.esquerda.max$ = 23 é maior que $i.baixo$ = 22, o laço continua com x como o filho à esquerda da raiz — o nó que contém [8, 9], que também não se sobrepõe a i. Dessa vez, $x.esquerda.max$ = 10 é menor que $i.baixo$ = 22 e, assim, o laço continua com o filho à direita de x como o novo x. Como o intervalo [15, 23] armazenado nesse nó se sobrepõe a i, o procedimento retorna esse nó.

Como exemplo de uma busca malsucedida, suponha que desejemos encontrar um intervalo que se sobreponha a i = [11, 14] na árvore de intervalos da Figura 14.4. Mais uma vez, começamos com x como a raiz. Visto que o intervalo [16, 21] da raiz não se sobrepõe a i, e visto que $x.esquerda.max$ = 23 é maior que $i.baixo$ = 11, vamos para a esquerda até o nó que contém [8, 9]. O intervalo [8, 9] não se sobrepõe a i e $x.esquerda.max$ = 10 é menos que $i.baixo$ = 11 e, portanto, vamos para a direita. (Observe que nenhum intervalo na subárvore à esquerda se sobrepõe a i.) O intervalo [15, 23] não se sobrepõe a i, e seu filho à esquerda é $T.nil$; assim, vamos novamente para a direita, o laço termina e retornamos a sentinela $T.nil$.

Para ver por que INTERVAL-SEARCH é correto, devemos entender por que basta examinar um único caminho simples que parte da raiz. A ideia básica é que em qualquer nó x, se $x.int$ não se sobrepõe a i, a busca sempre prossegue em uma direção segura: a busca definitivamente encontrará um intervalo sobreposto se a árvore contiver algum. O teorema a seguir enuncia essa propriedade de modo mais preciso.

Teorema 14.2

Qualquer execução de INTERVAL-SEARCH(T, i) devolve um nó cujo intervalo se sobrepõe a i ou devolve $T.nil$ e a árvore T não contém nenhum nó cujo intervalo se sobrepõe a i.

Prova O laço **while** das linhas 2 a 5 termina quando $x = T.nil$ ou i se sobrepõe a $x.int$. Nesse último caso, certamente é correto devolver x. Portanto, vamos focalizar o primeiro caso, no qual o laço **while** termina porque $x = T.nil$.

Usamos o seguinte invariante para o laço **while** das linhas 2 a 5:

Se a árvore T contém um intervalo que se sobrepõe a i, então a subárvore com raiz em x contém tal intervalo.

Figura 14.5 Intervalos na prova do Teorema 14.2. O valor de *x.esquerda.max* é mostrado em cada caso como uma linha tracejada. (**a**) A busca vai para a direita. Nenhum intervalo *i*' na subárvore à esquerda de *x* pode se sobrepor a *i*. (**b**) A busca vai para a esquerda. A subárvore à esquerda de *x* contém um intervalo que se sobrepõe a *i* (situação não mostrada) ou a árvore à esquerda de *x* contém um intervalo *i*' tal que *i*'*.alto* = *x.esquerda.max*. Visto que *i* não se sobrepõe a *i*', também não se sobrepõe a nenhum intervalo *i* "na subárvore à direita de *x*, já que *i*'*.baixo* ≤ *i*"*.baixo*.

Usamos esse invariante de laço da seguinte maneira:

Inicialização: Antes da primeira iteração, a linha 1 atribui a raiz de *T* a *x*, de modo que o invariante é válido.

Manutenção: Cada iteração do laço **while** executa a linha 4 ou a linha 5. Mostraremos que ambos os casos mantêm o invariante de laço.

Se a linha 5 é executada em razão da condição de desvio na linha 3, temos *x.esquerda* = *T.nil* ou *x.esquerda.max* < *i.baixo*. Se *x.esquerda* = *T.nil*, a subárvore com raiz em *x.esquerda* claramente não contém nenhum intervalo que se sobreponha a *i* e, assim, definir *x* como *x.direita* mantém o invariante. Portanto, suponha que *x.esquerda* ≠ *T.nil* e *x.esquerda.max* < *i.baixo*. Como mostra a Figura 14.5, para cada intervalo *i'* na subárvore à esquerda de *x*, temos

$$\begin{aligned} i'.alto &\leq x.esquerda.max \\ &< i.baixo \end{aligned}$$

Portanto, pela tricotomia de intervalos, *i*' e *i* não se sobrepõem. Assim, a subárvore à esquerda de *x* não contém nenhum intervalo que se sobreponha a *i*, de modo que atribuir *x.direita* a *x* como mantém o invariante.

Se, por outro lado, a linha 4 for executada, mostraremos que o contrapositivo do invariante de laço é válido. Isto é, se a subárvore com raiz em *x.esquerda* não contiver nenhum intervalo que se sobreponha a *i*, não há nenhum intervalo em nenhum lugar da árvore que se sobreponha a *i*. Visto que a linha 4 é executada, em razão da condição de desvio na linha 3, temos *x.esquerda.max* ≥ *i.baixo*. Além disso, pela definição do atributo *max*, a subárvore esquerda de *x* deve conter algum intervalo *i*' tal que

$$\begin{aligned} i'.alto &= x.esquerda.max \\ &\geq i.baixo \end{aligned}$$

(A Figura 14.5(b) ilustra a situação.) Visto que *i* e *i*' não se sobrepõem, e como não é verdade que *i*'*.alto* < *i.baixo*, decorre pela tricotomia de intervalos que *i.alto* < *i*'*.baixo*. As árvores de intervalos são chaveadas nas extremidades baixas de intervalos e, assim, a propriedade de árvore de busca implica que, para qualquer intervalo *i*" na subárvore direita de *x*,

$$
\begin{aligned}
i.alto &< i'.baixo \\
&\leq i''.baixo\ .
\end{aligned}
$$

Pela tricotomia de intervalos, i e i'' não se sobrepõem. Concluímos que, independentemente de qualquer intervalo na subárvore esquerda de x se sobrepor ou não a i, atribuir $x.esquerda$ a x mantém o invariante.

Término: Se o laço termina quando $x = T.nil$, a subárvore com raiz em x não contém nenhum intervalo que se sobreponha a i. O contrapositivo do invariante de laço implica que T não contém nenhum intervalo que se sobreponha a i. Daí, é correto retornar $x = T.nil$.

Portanto, o procedimento INTERVAL-SEARCH funciona corretamente.

Exercícios

14.3-1 Escreva pseudocódigo para LEFT-ROTATE que opere em nós em uma árvore de intervalos e atualize os atributos *max* no tempo $O(1)$.

14.3-2 Reescreva o código para INTERVAL-SEARCH de modo que ele funcione adequadamente quando todos os intervalos são abertos.

14.3-3 Descreva um algoritmo eficiente que, dado um intervalo i, retorne um intervalo que se sobreponha a i que tenha o ponto extremo baixo mínimo ou $T.nil$ se aquele intervalo não existir.

14.3-4 Dada uma árvore de intervalos T e um intervalo i, descreva como produzir uma lista com todos os intervalos em T que se sobrepõem a i no tempo $O(\min(n, k \lg n))$, onde k é o número de intervalos na lista de saída. (*Sugestão:* Um método simples executa várias consultas, modificando a árvore entre as consultas. Um método ligeiramente mais complicado não modifica a árvore.)

14.3-5 Sugira modificações para os procedimentos de árvores de intervalos para suportar a nova operação INTERVAL-SEARCH-EXACTLY(T, i), onde T é uma árvore de intervalos e i é um intervalo. A operação deve retornar um ponteiro para um nó x em T tal que $x.int.\ baixo = i.baixo$ e $x.int.alto = i.alto$ ou $T.nil$ se T não contiver nenhum nó desse tipo. Todas as operações, incluindo INTERVAL-SEARCH-EXACTILY, devem ser executadas no tempo $O(\lg n)$ em uma árvore de n nós.

14.3-6 Mostre como manter um conjunto dinâmico Q de números que suporta a operação MIN-GAP, que dá a magnitude da diferença entre os dois números mais próximos em Q. Por exemplo, se $Q = \{1, 5, 9, 15, 18, 22\}$, então MIN-GAP($Q$) retorna $18 - 15 = 3$, já que 15 e 18 são os dois números mais próximos em Q. Maximize a eficiência das operações INSERT, DELETE, SEARCH e MIN-GAP, e analise seus tempos de execução.

14.3-7 ★ Normalmente, os bancos de dados de VLSI representam um circuito integrado como uma lista de retângulos. Suponha que os lados de cada retângulo estejam alinhados paralelamente aos eixos x e y, de modo que podemos representar cada um deles por suas coordenadas x e y mínima e máxima. Dê um algoritmo de tempo $O(n \lg n)$ para decidir se um conjunto de n retângulos assim representados contém ou não dois retângulos que se sobrepõem. Seu algoritmo não precisa informar todos os pares que se interceptam, mas deve informar que existe uma sobreposição se um retângulo cobrir inteiramente outro retângulo, ainda que as linhas de contorno não se interceptem. (*Sugestão*: Movimente uma linha "de varredura" pelo conjunto de retângulos.)

14-1 Ponto de sobreposição máxima

Suponha que desejemos manter o controle de um ***ponto de sobreposição máxima*** em um conjunto de intervalos — um ponto que tenha o maior número de intervalos no banco de dados que se sobreponham a ele.

a. Mostre que sempre existirá um ponto de sobreposição máxima que é uma extremidade de um dos segmentos.

b. Projete uma estrutura de dados que suporte eficientemente as operações INTERVAL--INSERT, INTERVAL-DELETE e FIND-POM, que retorna um ponto de sobreposição máxima. (*Sugestão:* Mantenha uma árvore vermelho-preto de todas as extremidades. Associe o valor +1 a cada ponto extremo esquerdo e associe o valor −1 a cada ponto extremo direito. Aumente cada nó da árvore com algumas informações extras para manter o ponto de sobreposição máxima.)

14-2 Permutação de Josephus

Definimos o **problema de Josephus** da seguinte maneira. Suponha que n pessoas formem um círculo e que temos um inteiro positivo $m \leq n$. Começando com uma primeira pessoa designada, prosseguimos em torno do círculo, retirando cada m-ésima pessoa. Depois que cada pessoa é retirada, a contagem continua em torno do círculo restante. Esse processo continua até retirarmos todas as n pessoas do círculo. A ordem em que as pessoas são retiradas do círculo define a ***permutação de Josephus*** **(n, m)** dos inteiros 1, 2,..., n. Por exemplo, a permutação de Josephus (7, 3) é (3, 6, 2, 7, 5, 1, 4).

a. Suponha que m seja uma constante. Descreva um algoritmo de tempo $O(n)$ que, dado um inteiro n, dê como saída a permutação de Josephus (n, m).

b. Suponha que m não seja uma constante. Descreva um algoritmo de tempo $O(n \lg n)$ que, dados os inteiros n e m, dê como saída a permutação de Josephus (n, m).

NOTAS DO CAPÍTULO

Em seu livro, Preparata e Shamos [282] descrevem várias árvores de intervalos que aparecem na literatura, citando o trabalho de H. Edelsbrunner (1980) e E. M. McCreight (1981). O livro detalha uma árvore de intervalos que, dado um banco de dados estático de n intervalos, nos permite enumerar, no tempo $O(k + \lg n)$, todos os k intervalos que se sobrepõem a um determinado intervalo de consulta.

Parte

TÉCNICAS AVANÇADAS DE PROJETO E ANÁLISE

INTRODUÇÃO

Esta parte focaliza três técnicas importantes para projeto e análise de algoritmos eficientes: programação dinâmica (Capítulo 15), algoritmos gulosos (Capítulo 16) e análise amortizada (Capítulo 17). Partes anteriores apresentaram outras técnicas de extensa aplicação, como divisão e conquista, aleatorização e solução de recorrências. As técnicas aqui apresentadas são um pouco mais sofisticadas, mas nos ajudam a abordar muitos problemas de computação. Os temas introduzidos nesta parte aparecerão mais adiante no livro.

A programação dinâmica se aplica tipicamente a problemas de otimização nos quais fazemos um conjunto de escolhas para chegar a uma solução ótima. Ao fazermos cada escolha, muitas vezes, surgem subproblemas do mesmo tipo do problema geral abordado. A programação dinâmica é efetiva quando um determinado subproblema pode surgir de mais de um conjunto parcial de escolhas; a técnica fundamental é armazenar a solução para cada um desses subproblemas, para usar caso, eles apareçam novamente. O Capítulo 15 mostra como essa ideia simples, às vezes, pode transformar algoritmos de tempo exponencial em algoritmos de tempo polinomial.

Como os algoritmos de programação dinâmica, os algoritmos gulosos em geral se aplicam a problemas de otimização nos quais fazemos diversas escolhas para chegar a uma solução ótima. A ideia de um algoritmo guloso é fazer cada escolha de uma maneira ótima local. Um exemplo simples é o troco: para minimizar o número de moedas necessárias para dar o troco para uma determinada quantia, podemos selecionar repetidamente a moeda de maior denominação (valor de face) que não seja maior que a quantia que resta. Uma abordagem gulosa dá uma solução ótima para muitos problemas desse tipo muito mais rapidamente do que uma abordagem de programação dinâmica daria. Porém, nem sempre é fácil dizer se uma abordagem gulosa será efetiva. O Capítulo 16 apresenta a teoria dos matroides, que dá uma base matemática que pode nos ajudar a mostrar que um algoritmo guloso produz uma solução ótima.

Usamos análise amortizada para analisar certos algoritmos que executam uma sequência de operações semelhantes. Em vez de estimar o custo da sequência de operações limitando o custo real de cada operação separadamente, uma análise amortizada impõe um limite para o custo real da sequência inteira. Uma vantagem dessa abordagem é que, embora algumas operações possam ser caras, muitas outras podem ser baratas. Em outras palavras, muitas das operações poderiam ser executadas em tempos bem menores que o tempo do pior caso. Porém, a análise amortizada não é apenas uma ferramenta de análise; é também um modo de pensar no projeto de algoritmos, já que o projeto de um algoritmo e a análise de seu tempo de execução muitas vezes estão intimamente relacionados. O Capítulo 17 apresenta três modos de executar uma análise amortizada de um algoritmo.

15 Programação dinâmica

A programação dinâmica, assim como o método de divisão e conquista, resolve problemas combinando as soluções para subproblemas. (Nesse contexto, "programação" se refere a um método tabular, não ao processo de escrever código de computador.) Como vimos nos Capítulos 2 e 4, os algoritmos de divisão e conquista subdividem o problema em subproblemas independentes, resolvem os subproblemas recursivamente e depois combinam suas soluções para resolver o problema original. Ao contrário, a programação dinâmica se aplica quando os subproblemas se sobrepõem, isto é, quando os subproblemas compartilham subsubproblemas. Nesse contexto, um algoritmo de divisão e conquista trabalha mais que o necessário, resolvendo repetidamente os subsubproblemas comuns. Um algoritmo de programação dinâmica resolve cada subsubproblema só uma vez e depois grava sua resposta em uma tabela, evitando assim, o trabalho de recalcular a resposta toda vez que resolver cada subsubproblema.

Em geral, aplicamos a programação dinâmica a ***problemas de otimização***. Tais problemas podem ter muitas soluções possíveis. Cada solução tem um valor, e desejamos encontrar uma solução com o valor ótimo (mínimo ou máximo). Denominamos tal solução "*uma*" solução ótima para o problema, ao contrário de "*a*" solução ótima, já que podem existir várias soluções que alcançam o valor ótimo.

O desenvolvimento de um algoritmo de programação dinâmica segue uma sequência de quatro etapas:

1. Caracterizar a estrutura de uma solução ótima.
2. Definir recursivamente o valor de uma solução ótima.
3. Calcular o valor de uma solução ótima, normalmente de baixo para cima.
4. Construir uma solução ótima com as informações calculadas.

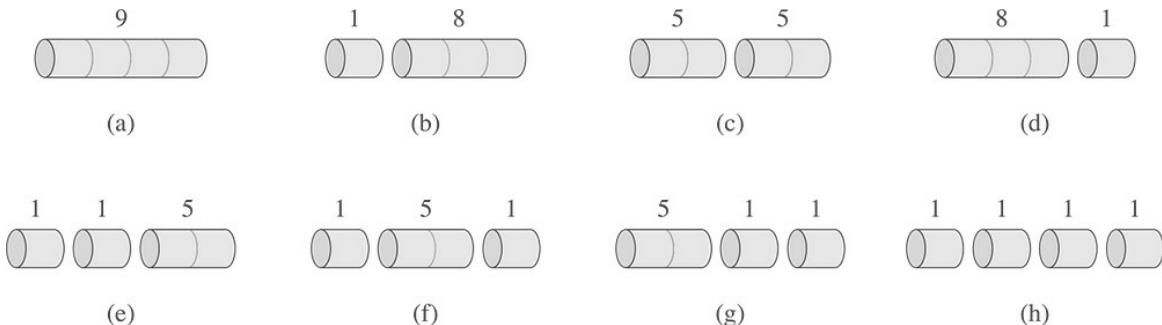
As etapas 1 a 3 formam a base de uma solução de programação dinâmica para um problema. Se precisarmos apenas do valor de uma solução ótima, e não da solução em si, podemos omitir a etapa 4. Quando executamos a etapa 4, às vezes mantemos informações adicionais durante a etapa 3, para facilitar a construção de uma solução ótima.

As seções a seguir, usam o método de programação dinâmica para resolver alguns problemas de otimização. A Seção 15.1 examina o problema de cortar uma haste em hastes de menor comprimento, de modo a maximizar os valores totais. A Seção 15.2 pergunta como podemos multiplicar uma cadeia de matrizes e, ao mesmo tempo, executar o menor número total de multiplicações escalares. Dados esses exemplos de programação dinâmica, a Seção 15.3 discute duas características fundamentais que um problema deve ter para que a programação dinâmica seja uma técnica de solução viável. Em seguida, a Seção 15.4 mostra como determinar a subsequência comum mais longa de duas sequências por programação dinâmica. Finalmente, a Seção 15.5 utiliza programação dinâmica para construir árvores de busca binária que são ótimas, dada uma distribuição conhecida de chaves que devem ser examinadas.

15.1 Corte de hastes

Nosso primeiro exemplo usa programação dinâmica para resolver um problema simples que é decidir onde cortar hastes de aço. A Serling Enterprises compra hastes de aço longas e as corta em hastes mais curtas, para vendê-las. Cada corte é livre. A gerência da Serling Enterprises quer saber qual é o melhor modo de cortar as hastes.

Suponhamos que conhecemos, para $i = 1, 2, ..., n$ o preço p_i em dólares que a Serling Enterprises cobra por uma haste de i polegadas de comprimento. Os comprimentos das hastes são sempre um número inteiro de polegadas. A Figura 15.1 apresenta uma amostra de tabela de preços.

O ***problema do corte de hastes*** é o seguinte. Dada uma haste de n polegadas de comprimento e uma tabela de preços p_i para $i = 1, 2, ..., n$, determine a receita máxima r_n que se pode obter cortando a haste e vendendo os pedaços. Observe que, se o preço p_n para uma haste de comprimento n for suficientemente grande, uma solução ótima pode exigir que ela não seja cortada.

Considere o caso quando $n = 4$. A Figura 15.2 mostra todas as maneiras de cortar uma haste de 4 polegadas de comprimento, incluindo não cortá-la. Vemos que cortar uma haste de 4 polegadas em duas peças de 2 polegadas produz a receita $p_2 + p_2 = 5 + 5 = 10$, que é ótima.

Podemos cortar uma haste de comprimento n de 2^{n-1} modos diferentes — já que temos uma opção independente de cortar ou não cortar — à distância de i polegadas da extremidade esquerda, para $i = 1, 2, ..., n - 1$.[1] Denotamos um desdobramento em pedaços usando notação de adição comum, portanto $7 = 2 + 2 + 3$ indica que uma haste de comprimento 7 foi cortada em três peças — duas de comprimento 2 e uma de comprimento 3. Se uma solução ótima cortar a haste em k pedaços, para algum $1 \le k \le n$, então o desdobramento ótimo

$$n = i_1 + i_2 + ... + i_k$$

comprimento i	1	2	3	4	5	6	7	8	9	10
preço p_i	1	5	8	9	10	17	17	20	24	30

Figura 15.1 Uma amostra de tabela de preços para hastes. Cada haste de comprimento i polegadas rende p_i dólares de receita para a empresa.

Figura 15.2 Os oito modos possíveis de cortar uma haste de comprimento 4. Acima de cada pedaço está o valor de cada um, de acordo com a amostra de tabela de preços na Figura 15.1. A estratégia ótima é a parte (c) — cortar a haste em duas peças de comprimento 2 — cujo valor total é 10.

da haste em peças de comprimentos $i_1, i_2, ..., i_k$ dá a receita máxima correspondente

$$r_n = p_{i_1} + p_{i_2} + ... + p_{i_k}.$$

Em nossa problema-amostra podemos determinar os números da receita ótima r_i , para $i = 1, 2, ..., 10$, por inspeção, com os correspondentes desdobramentos ótimos

$r_1 = 1$ pela solução 1 = 1 (nenhum corte),
$r_2 = 5$ pela solução 2 = 2 (nenhum corte),
$r_3 = 8$ pela solução 3 = 3 (nenhum corte),
$r_4 = 10$ pela solução 4 = 2 + 2,
$r_5 = 13$ pela solução 5 = 2 + 3,
$r_6 = 17$ pela solução 6 = 6 (nenhum corte),
$r_7 = 18$ pela solução 7 = 1 + 6 ou 7 + 2 + 2 + 3,
$r_8 = 22$ pela solução 8 = 2 + 6,
$r_9 = 25$ pela solução 9 = 3 + 6,
$r_{10} = 30$ pela solução 10 = 10 (nenhum corte).

De modo mais geral, podemos enquadrar os valores r_n para $n \geq 1$ em termos de receitas ótimas advindas de hastes mais curtas:

$$r_n = \max (p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1) \qquad (15.1)$$

O primeiro argumento, p_n, corresponde a não fazer nenhum corte e vender a haste de comprimento n como tal. Os outros $n - 1$ argumentos para max correspondem à receita máxima obtida de um corte inicial da haste em duas peças de tamanhos i e $n - i$, para cada $i = 1, 2, ...,n - 1$ e, prosseguindo com o corte ótimo dessas peças, obtendo as receitas r_i e r_{n-i} dessas duas peças. Visto que não sabemos de antemão qual é o valor de i que otimiza a receita, temos de considerar todos os valores possíveis para i e escolher aquele que maximize a receita. Temos também a opção de não escolher nenhum i se pudermos obter mais receita vendendo a haste inteira.

Observe que, para resolver o problema original de tamanho n, resolvemos problemas menores do mesmo tipo, porém de tamanhos menores. Uma vez executado o primeiro corte, podemos considerar os dois pedaços como instâncias independentes do problema do corte da haste. A solução ótima global incorpora soluções ótimas para os dois subproblemas relacionados, maximizando a receita gerada por esses dois pedaços. Dizemos que o problema do corte de hastes exibe ***subestrutura ótima***: soluções ótimas para um problema incorporam soluções ótimas para subproblemas relacionados, que podemos resolver independentemente.

Um modo relacionado, mas ligeiramente mais simples de arranjar uma estrutura recursiva para o problema do corte de hastes considera que um desdobramento consiste em uma primeira peça de comprimento i cortada da extremidade esquerda e o que restou do lado direito, com comprimento $n - i$. Somente o resto, e não a primeira peça, pode continuar a ser dividido. Podemos considerar cada desdobramento de uma haste de comprimento n desse modo: uma primeira peça seguida por algum desdobramento do resto. Quando fazemos isso, podemos expressar a solução que não contém nenhum corte dizendo que a primeira peça tem tamanho $i = n$ e receita p_n e que o resto tem tamanho 0 com receita correspondente $r_0 = 0$. Assim, obtemos a seguinte versão mais simples da equação (15.1):

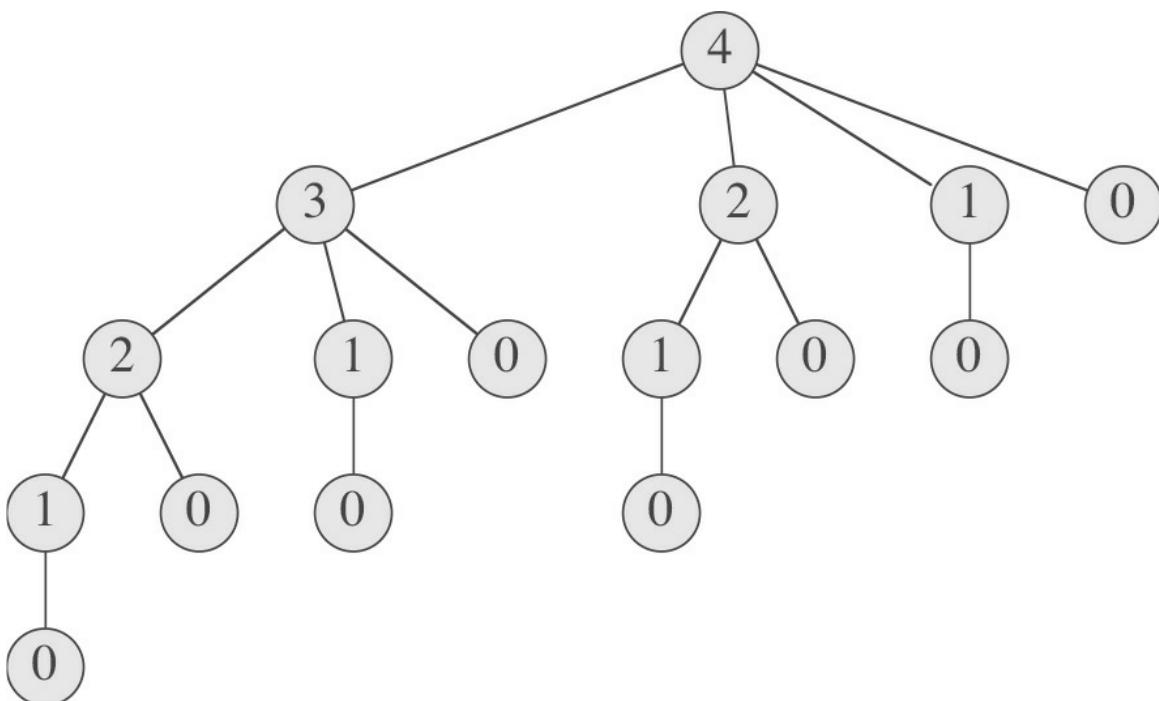
$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}). \qquad (15.2)$$

Nessa formulação, uma solução ótima incorpora a solução para somente *um* subproblema relacionado — o resto — em vez de dois.

Implementação recursiva de cima para baixo

O procedimento a seguir, implementa o cálculo implícito na equação (15.2) de um modo direto, recursivo, de cima para baixo.

```
CUT-ROD(p, n)
1  if n == 0
2     return 0
3  q = -∞
4  for i = 1 to n
5     q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

O procedimento CUT-ROD toma como entrada um arranjo $p[1 \,..n]$ de preços e um inteiro n, e retorna a máxima receita possível para uma haste de comprimento n. Se $n = 0$, nenhuma receita é possível e, portanto, CUT-ROD retorna 0 na linha 2. A linha 3 inicializa a receita máxima q para $-\infty$, de modo que o laço **for** na linhas 4-5 calcula corretamente $q = \max_{1 \leq i \leq n}(p_i + \text{CUT-ROD}(p, n, i))$; então, a linha 6 retorna esse valor. Uma simples indução em n prova que essa resposta é igual à resposta desejada r_n, dada pela equação (15.2).

Se você tivesse de codificar CUT-ROD em sua linguagem de programação favorita e executá-lo em seu computador, veria que, tão logo o tamanho se torna moderadamente grande, seu programa levaria um longo tempo para executá-lo. Para $n = 40$, você verificaria que seu programa demoraria no mínimo vários minutos e, o que é mais provável, mais de uma hora. Na verdade, constataria que cada vez que n aumentasse de 1, o tempo de execução de seu programa seria aproximadamente duas vezes maior.

Por que CUT-ROD é tão ineficiente? O problema é que CUT-ROD chama a si mesmo recursivamente repetidas vezes com os mesmos valores de parâmetros; resolve os mesmos problemas repetidamente. A Figura 15.3 ilustra o que acontece para $n = 4$:

CUT-ROD(p, n) chama CUT-ROD$(p, n - i)$ para $i = 1, 2, ..., n$. Equivalentemente, CUT-ROD (p, n) chama CUT-ROD(p, j) para cada $j = 0, 1,..., n - 1$. Quando esse processo se desenrola recursivamente, a quantidade de trabalho realizada, em função de n, cresce explosivamente.

Para analisar o tempo de execução de CUT-ROD, denotamos por $T(n)$ o número total de chamadas feitas a CUT-ROD se chamado quando seu segundo parâmetro é igual a n. Essa expressão é igual ao número de nós em uma subárvore cuja raiz é identificada por n na árvore de recursão. A contagem inclui a chamada inicial à raiz. Assim, $T(0) = 1$ e

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j). \tag{15.3}$$

Figura 15.3 A árvore de recursão que mostra chamadas recursivas resultantes de uma chamada a CUT-ROD(p, n) para n = 4. Cada rótulo dá o tamanho n do subproblema correspondente, de modo que uma aresta de um pai com rótulo s para um filho com rótulo t corresponde a cortar uma peça inicial de tamanho s - t e deixar um subproblema restante de tamanho t. Um caminho da raiz a uma folha corresponde a um dos 2^{n-1} modos de cortar uma haste de comprimento n. Em geral, essa árvore de recursão tem 2^n nós e 2^{n-1} folhas.

O 1 inicial é para a chamada na raiz, e o termo $T(j)$ conta o número de chamadas (incluindo chamadas recursivas) resultantes da chamada CUT-ROD(p, $n - i$), onde $j = n - i$.

Como o Exercício 15.1-1 pede que você mostre,

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j). \tag{15.3}$$

e, assim, o tempo de execução de CUT-ROD é exponencial em n.

Em retrospectiva, esse tempo de execução exponencial não é surpresa. CUT-ROD considera explicitamente todos os 2^{n-1} modos possíveis de cortar uma haste de comprimento n. A árvore de chamadas recursivas tem 2^{n-1} folhas, uma para cada modo possível de cortar a haste. Os rótulos no caminho simples da raiz até uma folha dão os tamanhos de cada pedaço restante do lado direito da haste antes de cada corte. Isto é, os rótulos dão os pontos de corte correspondentes, medidos em relação à extremidade direita da haste.

Utilização de programação dinâmica para o corte ótimo de hastes

Agora, mostramos como converter CUT-ROD em um algoritmo eficiente usando programação dinâmica.

O método da programação dinâmica funciona da seguinte maneira: agora que já observamos que uma solução recursiva ingênua é ineficiente porque resolve os mesmos problemas repetidas vezes, nós a adaptamos para resolver cada problema somente *uma vez* e armazenar sua solução. Se precisarmos nos referir novamente a esse problema mais tarde, bastará que o examinemos, em vez de o recalcular. Assim, a programação dinâmica usa memória adicional para poupar tempo de computação; serve como exemplo de uma ***permuta tempo-memória.*** As economias de tempo podem ser espetaculares: uma solução de tempo exponencial pode ser transformada em uma solução de tempo polinomial. Uma abordagem de programação dinâmica é executada em tempo polinomial quando o número de problemas *distintos* envolvido é polinomial no tamanho da entrada e podemos resolver cada subproblema em tempo polinomial.

Normalmente, há dois modos equivalentes de implementar uma abordagem de programação dinâmica. Ilustraremos ambos com nosso exemplo do corte de hastes.

A primeira abordagem é ***de cima para baixo com memoização***.[2] Nessa abordagem, escrevemos o procedimento recursivamente de maneira natural, porém modificado para salvar o resultado de cada subproblema (normalmente em um arranjo ou tabela hash). Agora, o procedimento primeiro verifica se já resolveu anteriormente esse subproblema. Se já o resolveu, retorna o valor salvo, poupando computação adicional nesse nível; se ainda não o resolveu, o procedimento calcula o valor da maneira usual. Dizemos que o procedimento recursivo foi ***memoizado***; ele "lembra" quais resultados já calculou anteriormente.

A segunda abordagem é o ***método de baixo para cima***. Essa abordagem depende tipicamente de alguma noção natural do "tamanho" de um subproblema, tal que resolver qualquer subproblema particular depende somente de resolver subproblemas "menores". Ordenamos os subproblemas por tamanho e os resolvemos em ordem de tamanho, o menor primeiro. Ao resolvemos um determinado problema, já resolvemos todos os subproblemas menores dos quais sua solução depende, e já salvamos suas soluções. Resolvemos cada subproblema somente uma vez e, na primeira vez que o vemos, já estão resolvidos todos os seus subproblemas pré-requisitados.

Essas duas abordagens produzem algoritmos com o mesmo tempo de execução assintótico, exceto em circunstâncias incomuns, quando a abordagem de cima para baixo não executa recursão propriamente dita para

examinar todos os subproblemas possíveis. Muitas vezes, a abordagem de baixo para cima tem fatores constantes muito melhores, visto que tem menos sobrecarga para chamadas de procedimento.

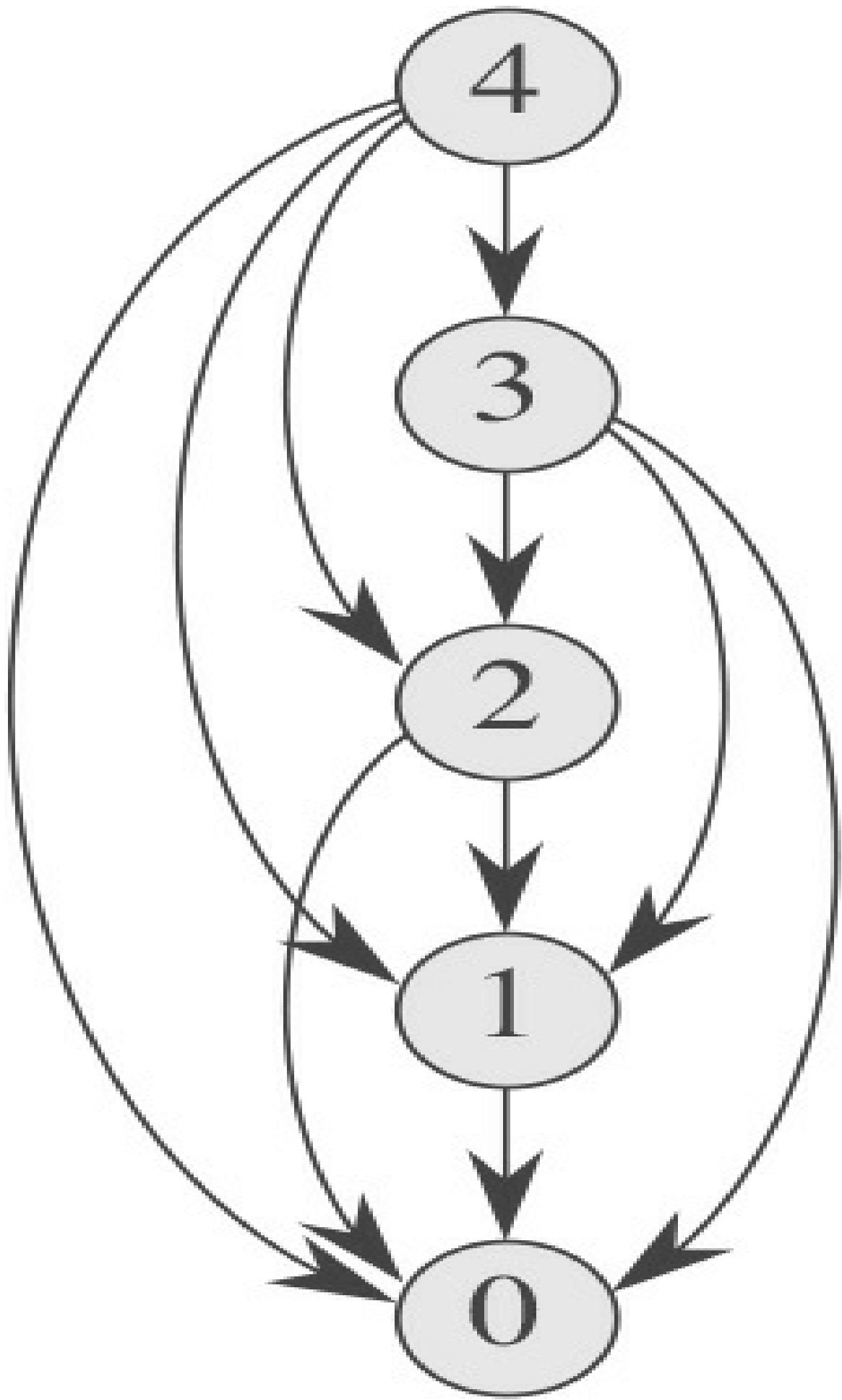
Apresentamos a seguir, o pseudocódigo para o procedimento CUT-ROD de cima para baixo, acrescido de memoização:

```
MEMOIZED-CUT-ROD(p, n)

1  seja r[0 .. n] um novo arranjo
2  for i = 0 to n
3     r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)

1  if r[n] ≥ 0
2  return r[n]
3  if n == 0
4     q = 0
5  else q = -∞
6     for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8     r[n] = q
9  return q
```

Aqui, o procedimento principal MEMOIZED-CUT-ROD inicializa um novo arranjo auxiliar $r[0 .. n]$ com o valor -∞, uma escolha conveniente para denotar "desconhecido". (Valores de receita conhecidos são sempre não negativos.) Então, ele chama sua rotina auxiliar, MEMOIZED-CUT-ROD-AUX.

O procedimento MEMOIZED-CUT-ROD-AUX é apenas a versão memoizada de nosso procedimento anterior, CUT-ROD. Primeiro ele consulta a linha 1 para verificar se o valor desejado já é conhecido; se for, a linha 2 retorna esse valor. Caso contrário, as linhas 3-7 calculam o valor desejado q na maneira usual, a linha 8 o salva em $r[n]$, e a linha 9 o retorna.

A versão de baixo para cima é ainda mais simples:

```
BOTTOM-UP-CUT-ROD(p, n)

1  seja r[0 .. n] um novo arranjo
2  r[0] = 0
3  for j = 1 to n
4     q = -∞
5     for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7     r[j] = q
8  return r[n]
```

No caso da abordagem da programação dinâmica de baixo para cima , BOTTOM-UP-CUT--ROD usa a ordenação natural dos subproblemas: um subproblema de tamanho i é "menor" do que um subproblema de tamanho j se $i < j$. Assim, o procedimento resolve subproblemas de tamanhos $j = 0, 1, ..., n$, naquela ordem.

A linha 1 do procedimento BOTTOM-UP-CUT-ROD cria um novo arranjo $r[0 .. n]$ no qual salvar os resultados dos subproblemas, e a linha 2 inicializa $r[0]$ com 0, visto que uma haste de comprimento 0 não rende nenhuma receita. As linhas 3-6 resolvem cada subproblema de tama>nho j, para $j = 1, 2, ..., n$, em ordem crescente de tamanho. A abordagem usada para resolver um problema de determinado tamanho j é a mesma usada por CUT-ROD, exceto que agora a linha 6 referencia diretamente a entrada $r[j - i]$ em vez de fazer uma chamada recursiva para resolver o subproblema de tamanho $j - i$. A linha 7 salva em $r[j]$ a solução do subproblema de tamanho j. Finalmente, a linha 8 retorna $r[n]$, que é igual ao valor ótimo r_n.

As versões de baixo para cima e de cima para baixo têm o mesmo tempo de execução assintótico. O tempo de execução do procedimento BOTTOM-UP-CUT-ROD é $Q(n_2)$, devido à sua estrutura de laço duplamente aninhado. O número de iterações de seu laço **for** interno nas linhas 5-6 forma uma série aritmética. O tempo de execução de sua contraparte de cima para baixo, MEMOIZED-CUT-ROD, também é $Q(n_2)$, embora esse tempo de execução possa ser um pouco mais difícil de ver. Como uma chamada recursiva para resolver um subproblema já resolvido antes retorna imediatamente, MEMOIZED-CUT-ROD resolve cada subproblema apenas uma vez. Esse procedimento resolve subproblemas para tamanhos 0, 1, ..., n. Para resolver um subproblema de tamanho n, o laço **for** das linhas 6-7 itera n vezes. Assim, o número total de iterações desse laço **for**, para todas as chamadas recursivas de MEMOIZED-CUT-ROD, forma uma série aritmética que dá um total de $Q(n_2)$ iterações, exatamente como o laço **for** interno de BOTTOM-UP-CUT-ROD. (Na verdade, aqui estamos usando uma forma de análise agregada. Veremos os detalhes da análise agregada na Seção 17.1.)

4
3
2
1
0

Figura 15.4 O grafo do subproblema para o problema do corte de haste com $n = 4$. Os rótulos nos vértices dão os tamanhos dos subproblemas correspondentes. Um vértice dirigido (x, y) indica que precisamos de uma solução para o subproblema y quando resolvemos o subproblema x. Esse grafo é uma versão reduzida da árvore da Figura 15.3, na qual todos os nós que têm o mesmo rótulo são integrados em um único vértice e todas as arestas vão de pai para filho.

Grafos de subproblemas

Quando pensamos em um problema de programação dinâmica, temos de entender o conjunto de subproblemas envolvido e como os subproblemas dependem uns dos outros.

O ***grafo de subproblemas*** para o problema incorpora exatamente essa informação. A Figura 15.4 mostra o grafo de subproblemas para o problema do corte de haste com $n = 4$. É um grafo dirigido, que contém um vértice para cada subproblema distinto. O grafo de subproblemas tem um vértice dirigido do vértice para o subproblema x até o vértice para o subproblema y se a determinação de uma solução ótima para o subproblema x envolver considerar diretamente uma solução ótima para o subproblema y. Por exemplo, o grafo de subproblema contém um vértice de x a y se um procedimento recursivo de cima para baixo para resolver x diretamente chamar a si mesmo para resolver y. Podemos imaginar o grafo de subproblema como uma versão "reduzida" ou "colapsada" da árvore de recursão para o método recursivo de cima para baixo, na qual reunimos todos os nós para o mesmo subproblema em um único vértice e orientamos todos os vértices de pai para filho.

O método de baixo para cima para programação dinâmica considera os vértices do grafo de subproblema em uma ordem tal que resolvemos os subproblemas y adjacentes a um dado subproblema x antes de resolvermos o subproblema x. (Lembre-se de que dissemos, na Seção B.4, que a relação de adjacência não é necessariamente simétrica.) Usando a terminologia do Capítulo 22, em um algoritmo de programação dinâmica de baixo para cima, consideramos os vértices do grafo de subproblema em uma ordem que é uma "ordenação topológica reversa" ou uma "ordenação topológica do transposto" (veja a Seção 22.4) do grafo de subproblema. Em outras palavras, nenhum subproblema é considerado até que todos os subproblemas dos quais ele depende tenham sido resolvidos. De modo semelhante, usando noções do mesmo capítulo, podemos considerar o método de cima para baixo (com memoização) para programação dinâmica como uma "busca em profundidade" do grafo de subproblema (veja a Seção 22.3).

O tamanho do grafo de subproblema $G = (V, E)$ pode nos ajudar a determinar o tempo de execução do algoritmo de programação dinâmica. Visto que resolvemos cada subproblema apenas uma vez, o tempo de execução é a soma dos tempos necessários para resolver cada subproblema. Normalmente, o tempo para calcular a solução de um subproblema é proporcional ao grau (número de vértices dirigidos para fora) do vértice correspondente no grafo de subproblema, e o número de subproblemas é igual ao número de vértices no grafo de subproblema. Nesse caso comum, o tempo de execução da programação dinâmica é linear em relação ao número de vértices e arestas.

Reconstruindo uma solução

Nossas soluções de programação dinâmica para o corte de hastes devolvem o valor de uma solução ótima, mas não devolvem uma solução propriamente dita: uma lista de tamanhos de peças. Podemos estender a abordagem da programação dinâmica para registrar não apenas o *valor* ótimo calculado para cada subproblema, mas também uma *escolha* que levou ao valor ótimo. Com essa informação, podemos imprimir imediatamente uma solução ótima.

Apresentamos a seguir uma versão estendida de BOTTOM-UP-CUT-ROD que calcula, para cada tamanho de haste j, não somente a receita máxima r_j, mas também s_j, o tamanho ótimo da primeira peça a ser cortada.

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
1   sejam r[0 .. n] e s[0 .. n] novos arranjos
2   r[0] = 0
3   for j = 1 to n
4       q = -∞
5       for i = 1 to j
6           if q < p[i] + r[j − i]
7               q = p[i] + r[j − i]
8               s[j] = i
9       r[j] = q
10  return r and s
```

Esse procedimento é semelhante a BOTTOM-UP-CUT-ROD, exceto pela criação do arranjo s na linha 1 e pela atualização de $s[j]$ na linha 8 para conter o tamanho ótimo i da primeira peça a cortar ao resolver um subproblema de tamanho j.

O procedimento apresentado a seguir toma um tabela de preços p e um tamanho de haste n e chama EXTENDED-BOTTOM-UP-CUT-ROD para calcular o arranjo $s[1 \; .. \; n]$ de tamanhos ótimos da primeira peça e depois imprime a lista completa de tamanho de peças conforme um desdobramento ótimo de uma haste de comprimento n:

PRINT-CUT-ROD-SOLUTION(p, n)

```
1   (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2   while n > 0
3       print s[n]
4       n = n − s[n]
```

Em nosso exemplo do corte de haste, a chamada EXTENDED-BOTTOM-UP-CUT-ROD$(p,10)$ retornaria os seguintes arranjos:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Uma chamada a PRINT-CUT-ROD-SOLUTION$(p,10)$ imprimiria apenas 10, mas uma chamada com $n = 7$ imprimiria os cortes 1 e 6, correspondentes ao primeiro desdobramento ótimo para r_7 dado anteriormente.

Exercícios

15.1-1 Mostre que a equação (15.4) decorre da equação (15.3) e da condição inicial $T(0) = 1$.

15.1-2 Mostre, por meio de um contraexemplo, que a seguinte estratégia "gulosa" nem sempre determina um modo ótimo de cortar hastes. Defina a ***densidade*** de uma haste de comprimento i como p_i/i, isto é, seu valor por polegada. A estratégia gulosa para uma haste de comprimento n corta uma primeira peça de comprimento i, onde $1 \leq i \leq n$, om densidade máxima. Então, continua aplicando a estratégia gulosa à peça resultante de comprimento $n - i$.

15.1-3 Considere uma modificação do problema do corte da haste no qual, além de um preço p_i para cada haste, cada corte incorre em um custo fixo c. A receita associada à solução agora é a soma dos preços das peças menos os custos da execução dos cortes. Dê um algoritmo de programação dinâmica para resolver esse problema modificado.

15.1-4 Modifique MEMOIZED-CUT-ROD para retornar não somente o valor, mas também a solução propriamente dita.

15.1-5 Os números de Fibonacci são definidos pela recorrência (3.22). Dê um algoritmo de programação dinâmica de tempo $O[n]$ para calcular o n–ésimo número de Fibonacci. Desenhe o grafo de subproblema. Quantos vértices e arestas existem no grafo?

15.2 MULTIPLICAÇÃO DE CADEIAS DE MATRIZES

Nosso próximo exemplo de programação dinâmica é um algoritmo que resolve o problema de multiplicação de cadeias de matrizes. Temos uma sequência (cadeia) $\langle A_1, A_2, ..., A_n \rangle$ de n matrizes para multiplicar e desejamos calcular o produto

$$A_1, A_2 \cdots A_n . \tag{15.5}$$

Podemos avaliar a expressão (15.5) usando o algoritmo-padrão para multiplicação de pares de matrizes como uma sub-rotina, tão logo a tenhamos parentizado para resolver todas as ambiguidades relativas à multiplicação das matrizes entre si. A multiplicação de matrizes é associativa e, portanto, não importa como são colocadas entre parênteses; o produto entre elas é sempre o mesmo. Um produto de matrizes é ***totalmente parentizado*** se for uma única matriz ou o produto de dois produtos de matrizes totalmente parentizadas também expresso entre parênteses. Por exemplo, se a cadeia de matrizes é $\langle A_1, A_2, A_3, A_4 \rangle$, podemos expressar o produto $A_1 A_2 A_3 A_4$ como totalmente parentizado de cinco modos distintos:

$$(A_1(A_2(A_3A_4))) ,$$
$$(A_1((A_2 A_3)A_4)) ,$$
$$((A_1 A_2)(A_3A_4)) ,$$
$$((A_1(A_2 A_3))A_4) ,$$
$$(((A_1 A_2)A_3)A_4) .$$

O modo como colocamos parênteses em uma cadeia de matrizes pode ter um impacto expressivo sobre o custo de avaliação do produto. Considere primeiro o custo de multiplicar duas matrizes. O algoritmo-padrão é dado pelo pseudocódigo a seguir, que generaliza o procedimento SQUARE-MATRIX-MULTIPLY da Seção 4.2. Os atributos *linhas* e *colunas* são os números de linhas e colunas em uma matriz.

```
MATRIX-MULTIPLY(A,B)
1  if A.colunas ≠ B.linhas
2      error "dimensões incompatíveis"
3  else seja C uma nova A.linhas • B.colunas
4      for i = 0 to A.linhas
5          for j = 1 to B.colunas
6              c_ij = 0
7              for k = 1 to A.colunas
8                  c_ij = c_ij + a_ik · a_jk
9      return C
```

Podemos multiplicar duas matrizes A e B somente se elas forem ***compatíveis***: o número de colunas de A deve ser igual ao número de linhas de B. Se A é uma matriz $p \cdot q$ e B é uma matriz $q \cdot r$, a matriz resultante C é uma matriz $p \cdot r$. O tempo para calcular C é dominado pelo número de multiplicações escalares na linha 8, que é pqr. A seguir, expressaremos os custos em termos do número de multiplicações escalares.

Para ilustrar os diferentes custos incorridos pelas diferentes posições dos parênteses em um produto de matrizes, considere o problema de uma cadeia $\langle A_1, A_2, A_3\rangle$ de três matrizes. Suponha que as dimensões das matrizes sejam 10 · 100, 100 · 5 e 5 · 50, respectivamente. Se multiplicarmos as matrizes de acordo com a posição dos parênteses $((A_1A_2)A_3)$, executaremos 10 · 100 · 5 = 5.000 multiplicações escalares para calcular o produto 10 · 5 de matrizes A_1 A_2, mais outras 10 · 5 · 50 = 2.500 multiplicações escalares para multiplicar essa matriz por A_3, produzindo um total de 7.500 multiplicações escalares. Se, em vez disso, multiplicarmos de acordo com a posição dos parênteses $(A_1(A_2\ A_3))$, executaremos 100 · 5 · 50 = 25.000 multiplicações escalares para calcular o produto 100 · 50 de matrizes $A_2\,A_3$, mais outras 10 · 100 · 50 = 50.000 multiplicações escalares para multiplicar A_1 por essa matriz, dando um total de 75.000 multiplicações escalares. Assim, o cálculo do produto de acordo com a primeira posição dos parênteses é 10 vezes mais rápido.

Enunciamos o ***problema de multiplicação de cadeias de matrizes*** da seguinte maneira: dada uma cadeia $\langle A_1, A_2, \ldots A_n\rangle$ de n matrizes onde, para $i = 1, 2, \ldots, n$, a matriz A_i tem dimensão $p_{i-1} \cdot p_i$, expresse o produto $A_1, A_2, \ldots A_n$ como um produto totalmente parentizado de modo a minimizar o número de multiplicações escalares.

Observe que no problema de multiplicação de cadeias de matrizes, não estamos realmente multiplicando matrizes. Nossa meta é apenas determinar uma ordem para multiplicar matrizes que tenha o custo mais baixo. Em geral, o tempo investido na determinação dessa ordem ótima é mais que compensado pelo tempo economizado mais tarde, quando as multiplicações de matrizes são de fato executadas (por exemplo, executar apenas 7.500 multiplicações escalares em vez de 75.000).

Contagem do número de parentizações

Antes de resolver o problema de multiplicação de cadeias de matrizes por programação dinâmica, temos de nos convencer de que a verificação exaustiva de todas as possíveis parentizações não resulta em um algoritmo eficiente. Vamos representar por $P(n)$ o número de parentizações alternativas de uma sequência de n matrizes. Quando $n = 1$, há apenas uma matriz e, portanto, somente um modo de parentizar totalmente o produto de matrizes. Quando $n = 2$, um produto de matrizes totalmente parentizado é o produto de dois subprodutos de matrizes totalmente parentizados, e a separação entre os dois subprodutos pode ocorrer entre a k-ésima e a $(k + 1)$-ésima matrizes para qualquer $k = 1, 2, \ldots, n - 1$. Assim, obtemos a recorrência

$$P(n) = \begin{cases} 1 & \text{se } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{se } n \geq 2. \end{cases} \quad (15.6)$$

O Problema 12-4 pediu para mostrar que a solução para uma recorrência semelhante é a sequência de ***números de Catalan***, que cresce como $(4^n/n_{3/2})$. Um exercício mais simples (veja o Exercício 15.2-3) é mostrar que a solução para a recorrência (15.6) é (2^n). Portanto, o número de soluções é exponencial em n e o método de força bruta de busca exaustiva é uma estratégia ruim para determinar a parentização ótima de uma cadeia de matrizes.

Aplicação da programação dinâmica

Usaremos o método da programação dinâmica para determinar a parentização ótima de uma cadeia de matrizes. Para tal, seguiremos a sequência de quatro etapas declaradas no início deste capítulo:

1. Caracterizar a estrutura de uma solução ótima.
2. Definir recursivamente o valor de uma solução ótima.
3. Calcular o valor de uma solução ótima.
4. Construir uma solução ótima com as informações calculadas.

Percorreremos essas etapas em ordem, demonstrando claramente como aplicar cada etapa ao problema.

Etapa 1: A estrutura de uma parentização ótima

Em nossa primeira etapa do paradigma de programação dinâmica, determinamos a subestrutura ótima e depois a usamos para construir uma solução ótima para o problema partindo de soluções ótimas para subproblemas. No problema de multiplicação de cadeias de matrizes, podemos executar essa etapa da maneira descrita a seguir. Por conveniência, vamos adotar a notação $A_{i..j}$, onde $i \leq j$ para a matriz que resulta da avaliação do produto $A_i A_{i+1} \cdots A_j$. Observe que, se o problema é não trivial, isto é, se $i < j$, então, para parentizar o produto $A_i A_{i+1} \cdots A_j$ temos de separá-lo entre A_k e A_{k+1} para algum inteiro k no intervalo $i \leq k < j$. Isto é, para algum valor de k, primeiro calculamos as matrizes $A_{i..k}$ e $A_{k+1..j}$, e depois multiplicamos as duas para gerar o produto final $A_{i..j}$. O custo dessa parentização é o custo de calcular a matriz $A_{i..k}$, mais o custo de calcular $A_{k+1..j}$, mais o custo de multiplicá-las uma pela outra.

A subestrutura ótima desse problema é dada a seguir. Suponha que para efetuar a parentização ótima de $A_i A_{i+1} \cdots A_j$ separamos o produto entre A_k e A_{k+1}. Então, o modo como posicionamos os parênteses na subcadeia "prefixo" $A_i A_{i+1} \cdots A_k$ dentro dessa parentização ótima de $A_i A_{i+1} \cdots A_j$ deve ser uma parentização ótima de $A_i A_{i+1} \cdots A_k$. Por quê? Se existisse um modo menos dispendioso de parentizar $A_i A_{i+1} \cdots A_k$, então poderíamos substituir essa parentização na parentização ótima de $A_i A_{i+1} \cdots A_j$ para produzir um outro modo de parentizar $A_i A_{i+1} \cdots A_j$ cujo custo seria mais baixo que o custo ótimo: uma contradição. Uma observação semelhante é válida para parentizar a subcadeia $A_{k+1} A_{k+2} \cdots A_j$ na parentização ótima de $A_i A_{i+1} \cdots A_j$: ela deve ser uma parentização ótima de $A_{k+1} A_{k+2} \cdots A_j$.

Agora, usamos nossa subestrutura ótima para mostrar que podemos construir uma solução ótima para o problema pelas soluções ótimas para subproblemas. Vimos que qualquer solução para uma instância não trivial do problema de multiplicação de cadeias de matrizes requer que separemos o produto e que qualquer solução ótima contém em si soluções ótimas para instâncias de subproblemas. Assim, podemos construir uma solução ótima para uma instância do problema de multiplicação de cadeias de matrizes separando o problema em dois subproblemas (pela parentização ótima de $A_i A_{i+1} \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$), determinando soluções ótimas para instâncias de subproblemas e depois combinando essas soluções ótimas de subproblemas. Devemos assegurar que, quando procurarmos o lugar correto para separar o produto, consideremos todos os lugares possíveis, para termos certeza de que examinamos a opção ótima.

Etapa 2: Uma solução recursiva

Em seguida, definimos recursivamente o custo de uma solução ótima em termos das soluções ótimas para subproblemas. No caso do problema de multiplicação de cadeias de matrizes, escolhemos como nossos subproblemas os problemas da determinação do custo mínimo da parentização de $A_i A_{i+1} \cdots A_j$ para $1 \leq i \leq j \leq n$. Seja $m[i, j]$ o número mínimo de multiplicações escalares necessárias para calcular a matriz $A_{i..j}$; para o problema completo, o custo mínimo para calcular $A_{1..n}$ seria, portanto, $m[1, n]$.

Podemos definir $m[i, j]$ recursivamente da maneira descrita a seguir. Se $i = j$, o problema é trivial; a cadeia consiste em apenas uma matriz $A_{i..i} = A_i$, de modo que nenhuma multiplicação escalar é necessária para calcular o produto. Assim, $m[i, i] = 0$ para $i = 1, 2, ..., n$. Para calcular $m[i, j]$ quando $i < j$, tiramos proveito da estrutura de uma solução ótima da etapa 1. Vamos considerar que, para obter a parentização ótima, separamos o produto $A_i A_{i+1} \cdots A_j$ entre A_k e A_{k+1}, onde $i \leq k < j$. Então, $m[i, j]$ é igual ao custo mínimo para calcular os subprodutos $A_{i..k}$ e $A_{k+1..j}$, mais o custo de multiplicar essas duas matrizes. Recordando que cada matriz A_i é $p_{i-1} \cdot p_i$, vemos que o cálculo do produto de matrizes $A_{i..k} A_{k+1..j}$ exige $p_{i-1} p_k p_j$ multiplicações escalares. Assim, obtemos

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j .$$

Essa equação recursiva supõe que conhecemos o valor de k, o que não é verdade. Porém, há somente $j - i$ valores possíveis para k, isto é, $k = i, i + 1, ..., j - 1$. Visto que a parentização ótima deve usar um desses valores para k, precisamos apenas verificar todos eles para determinar o melhor. Assim, nossa definição recursiva para o custo mínimo de colocar o produto $A_i A_{i+1} \cdots A_j$ entre parênteses se torna

$$m[i, j] = \begin{cases} 0 & \text{se } i = j, \\ \min\limits_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{se } i < j. \end{cases} \quad (15.7)$$

Os valores $m[i, j]$ dão os custos de soluções ótimas para subproblemas, mas não dão todas as informações de que necessitamos para construir uma solução ótima. Para nos ajudar a fazer isso, definimos $s[i, j]$ como um valor de k no qual separamos o produto $A_i A_{i+1} \cdots A_j$ para obter uma parentização ótima. Isto é, $s[i, j]$ é igual a um valor k tal que $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Etapa 3: Cálculo dos custos ótimos

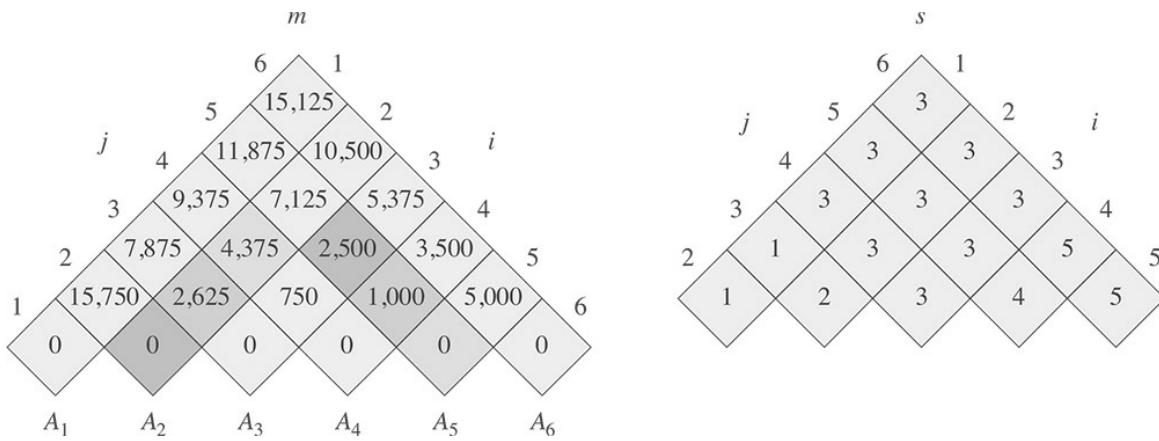
Neste ponto, seria fácil escrever um algoritmo recursivo baseado na recorrência (15.7) para calcular o custo mínimo $m[1, n]$ para multiplicar $A_1 A_2 \cdots A_n$. Como vimos no problema do corte de hastes de aço, e como veremos na Seção 15.3, esse algoritmo recursivo demora um tempo exponencial, o que não é nada melhor que o método da força bruta de verificar cada maneira de parentizar o produto.

Observe que temos um número relativamente pequeno de subproblemas distintos: um problema para cada escolha de i e j que satisfaça $1 \leq i \leq j \leq n$ ou $\frac{(n)}{(2)} + n = Q(n_2)$ no total. Um algoritmo recursivo pode encontrar cada subproblema, muitas vezes, em diferentes ramos de sua árvore de recursão. Essa propriedade de sobrepor subproblemas é a segunda indicação da aplicabilidade da programação dinâmica (a subestrutura ótima é a primeira).

Em vez de calcular a solução para a recorrência (15.7) recursivamente, calculamos o custo ótimo usando uma abordagem tabular de baixo para cima. (Na Seção 15.3, apresentaremos a abordagem de cima para baixo correspondente usando memoização.)

Implementaremos o método tabular, de cima para baixo, no procedimento MATRIX-CHAIN--ORDER, que aparece mais adiante. Esse procedimento supõe que a matriz A_i tem dimensões $p_{i-1} \cdot p_i$ para $i = 1, 2, ..., n$. Sua entrada é uma sequência $p = \langle p_0, p_1, ..., p_n \rangle$, onde *p.comprimento* $= n + 1$. O procedimento utiliza uma tabela auxiliar $m[1 .. n, 1 .. n]$

para armazenar os custos $m[i, j]$ e uma outra tabela auxiliar $s[1 .. n - 1 .. n]$ que registra qual índice de k alcançou o custo ótimo no cálculo de $m[i, j]$. Usaremos a tabela s para construir uma solução ótima.

Para implementar a abordagem de baixo para cima, devemos determinar a quais entradas da tabela nos referimos para calcular $m[i, j]$. A equação (15.7) mostra que o custo $m[i, j]$ de calcular um produto de cadeias de $j - i + 1$ matrizes só depende dos custos de calcular os produtos de cadeias de menos que $j - i + 1$ matrizes. Isto é, para $k = i, i + 1, ..., j - 1$, a matriz $A_{i..k}$ é um produto de $k - i + 1 < j - i + 1$ matrizes, e a matriz $A_{k+1..j}$ é um produto de $j - k < j - i + 1$ matrizes. Assim, o algoritmo deve preencher a tabela m de um modo que corresponda a resolver o problema da parentização em cadeias de matrizes de comprimento crescente. Para o subproblema da parentização ótima da cadeia $A_i A_{i+1} \cdots A_j$, admitimos que o tamanho do subproblema é o comprimento $j - i + 1$ da cadeia.

```
MATRIX-CHAIN-ORDER (p)

1   n = p.comprimento − 1
2   sejam m[1 .. n, 1 .. n] e s[1 .. n − 1,2 .. n] tabelas novas
3   for i = 1 to n
4      m[i, i] = 0
5   for l = 2 to n                  // l é o comprimento da cadeia
6      for i = 1 to n − l + 1
7          j = i + l − 1
8          m[i, j] = ∞
9          for k = i to j − 1
10             q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
11             if q < m[i, j]
12                 m[i, j] = q
13                 s[i, j] = k
14  return m e s
```

O algoritmo calcula primeiro $m[i, i] = 0$ para $i = 1, 2, ..., n$ (os custos mínimos para cadeias de comprimento 1) nas linhas 3-4. Então, usa a recorrência (15.7) para calcular $m[i, i + 1]$ para $i = 1, 2, ..., n - 1$ (os custos mínimos para cadeias de comprimento $l = 2$) durante a primeira execução do laço **for** nas linhas 5-13. Na segunda passagem pelo laço, o algoritmo calcula $m[i, i + 2]$ para $i = 1, 2, ..., n - 2$ (os custos mínimos para cadeias de comprimento $l = 3$), e assim por diante. Em cada etapa, o custo $m[i, j]$ calculado nas linhas 10-13 depende apenas das entradas de tabela $m[i, k]$ e $m[k + 1, j]$ já calculadas.

A Figura 15.5 ilustra esse procedimento em uma cadeia de $n = 6$ matrizes. Visto que definimos $m[i, j]$ somente para $i = j$, apenas a porção da tabela m estritamente acima da diagonal principal é usada. A tabela mostrada na figura sofreu uma rotação para colocar a diagonal principal na posição horizontal. A lista ao longo da parte inferior da figura mostra a cadeia de matrizes. Usando esse leiaute, podemos determinar o custo mínimo $m[i, j]$ para multiplicar uma subcadeia de matrizes $A_i A_{i+1} \cdots A_j$ na interseção de linhas que partem de A_i na direção nordeste e de A_j. Cada linha horizontal na tabela contém as entradas para cadeias de matrizes do mesmo comprimento. MATRIX-CHAIN-ORDER calcula as linhas de baixo para cima e da esquerda para a direita dentro de cada linha. Calcula cada entrada $m[i, j]$ usando os produtos $p_{i-1} p_k p_j$ para $k = i, i + 1, ..., j - 1$ e todas as entradas a sudoeste e a sudeste de $m[i, j]$.

Uma simples inspeção da estrutura de laços aninhados de MATRIX-CHAIN-ORDER produz um tempo de execução de $O(n_3)$ para o algoritmo. Os laços estão aninhados com profundidade três, e cada índice de laço (l, i e k) adota no máximo n valores. O Exercício 15.2-5 pede que você mostre que o tempo de execução desse algoritmo é mesmo (n_3). O algoritmo requer espaço $Q(n_2)$ para armazenar as tabelas m e s. Assim, MATRIX-CHAIN-ORDER é muito mais eficiente que o método de tempo exponencial que enumera todas as possíveis parentizações e verifica cada uma delas.

Figura 15.5 As tabelas m e s calculadas por MATRIX-CHAIN-ORDER para $n = 6$ e as seguintes dimensões de matrizes:

matriz	A_1	A_2	A_3	A_4	A_5	A_6
dimensão	$30 \cdot 35$	$35 \cdot 15$	$15 \cdot 5$	$5 \cdot 10$	$10 \cdot 20$	$20 \cdot 25$

As tabelas sofreram uma rotação para colocar a diagonal principal na posição horizontal. A tabela m usa somente a diagonal principal e o triângulo superior, e a tabela s usa somente o triângulo superior. O número mínimo de multiplicações escalares para multiplicar as seis matrizes é $m[1, 6] = 15.125$. Entre as entradas sombreadas nos três tons mais escuros, os pares que têm o mesmo sombreado são tomados juntos na linha 10 quando se calcula

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125.$$

Etapa 4: Construção de uma solução ótima

Embora determine o número ótimo de multiplicações escalares necessárias para calcular um produto de cadeias de matrizes, MATRIX-CHAIN-ORDER não mostra diretamente como multiplicar as matrizes. A tabela $s[1 .. n - 1, 2, ..., n]$ nos dá a informação que precisamos para fazer isso. Cada entrada $s[i, j]$ registra um valor de k tal que uma parentização ótima de $A_i A_i{+1} \cdots A_j$ separa o produto entre A_k e $A_k{+1}$. Assim, sabemos que a multiplicação de matrizes final no cálculo ótimo de $A_{1..n}$, é $A_{1 \cdot\cdot s1, n} A_{s1, n + 1 \cdot\cdot n}$. Podemos determinar as multiplicações de matrizes anteriores recursivamente, já que $s[1, s[1, n$ determina a última multiplicação de matrizes no cálculo de $A_{1 \cdot\cdot s1, n}$ e $s[s[1, n] + 1, n]$ determina a última multiplicação de matrizes no cálculo de $A_{s1, n+1 \cdot\cdot n}$. O procedimento recursivo a seguir imprime uma parentização ótima de $\langle A_i, A_i{+1}, ..., A_j \rangle$, dada a tabela s calculada por MATRIX-CHAIN-ORDER e os índices i e j. A chamada inicial PRINT-OPTIMAL-PARENS$(s, 1, n)$ imprime uma parentização ótima de $\langle A_1, A_2, ..., An \rangle$.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

No exemplo da Figura 15.5 a chamada PRINT-OPTIMAL-PARENS(s, 1, 6) imprime a parentização de $((A_1\ (A_2\ A_3))\ ((A_4\ A_5)\ A_6))$.

Exercícios

15.2-1 Determine uma parentização ótima de um produto de cadeias de matrizes cuja sequência de dimensões é ⟨5, 10, 3, 12, 5, 50, 6⟩.

15.2-2 Dê um algoritmo recursivo MATRIX-CHAIN-MULTIPLY(A, s, i, j) que realmente execute a multiplicação ótima de cadeias de matrizes, dadas a sequência de matrizes $\langle A_1, A_2, ..., A_n\rangle$, a tabela s calculada por MATRIX-CHAIN-ORDER e os índices i e j. (A chamada inicial seria MATRIX-CHAIN-MULTIPLY(A, s, 1, n).)

15.2-3 Use o método de substituição para mostrar que a solução para a recorrência (15.6) é $\Omega(2n)$.

15.2-4 Descreva o grafo de subproblema para multiplicação de cadeia de matrizes com uma cadeia de entrada de comprimento n. Quantos vértices ele tem? Quantas arestas ele tem e quais são essas arestas?

15.2-5 Seja $R(i, j)$ o número de vezes que a entrada de tabela $m[i, j]$ é referenciada durante o cálculo de outras entradas de tabela em uma chamada de MATRIX-CHAIN-ORDER. Mostre que o número total de referências para a tabela inteira é

$$\sum_{i=j}^{n}\sum_{j=i}^{n} R(i,j) = \frac{n^3 - n}{3}.$$

(*Sugestão*: A equação (A.3) pode ser útil.)

15.2-6 Mostre que uma parentização completa de uma expressão de n elementos tem exatamente n – 1 pares de parênteses.

15.3 ELEMENTOS DE PROGRAMAÇÃO DINÂMICA

Embora tenhamos acabado de analisar dois exemplos do método de programação dinâmica, é bem possível que você ainda esteja imaginando exatamente quando aplicar o método. Do ponto de vista da engenharia, quando devemos procurar uma solução de programação dinâmica para um problema? Nesta seção, examinamos os dois elementos fundamentais que um problema de otimização deve ter para que a programação dinâmica seja aplicável: subestrutura ótima e sobreposição de subproblemas. Também voltaremos a discutir mais completamente como a memoização pode

nos ajudar a aproveitar a propriedade de sobreposição de subproblemas em uma abordagem recursiva de cima para baixo.

Subestrutura ótima

O primeiro passo para resolver um problema de otimização por programação dinâmica é caracterizar a estrutura de uma solução ótima. Lembramos que um problema apresenta uma ***subestrutura ótima*** se uma solução ótima para o problema contiver soluções ótimas para subproblemas. Sempre que um problema exibir subestrutura ótima, temos uma boa indicação de que a programação dinâmica pode se aplicar. (Porém, como discutiremos no Capítulo 16, isso também pode significar que uma estratégia gulosa é aplicável.) Em programação dinâmica, construímos uma solução ótima para o problema partindo de soluções ótimas para subproblemas. Consequentemente, devemos ter o cuidado de garantir que a faixa de subproblemas que consideramos inclui aqueles usados em uma solução ótima.

Encontramos uma subestrutura ótima em ambos os problemas examinados neste capítulo até agora. Na Seção 15.1, observamos que o modo mais rápido de cortar uma haste de comprimento n (se a cortarmos) envolve cortar otimamente as duas peças resultantes do primeiro corte. Na Seção 15.2, observamos que uma parentização ótima de $A_i A_{i+1} \cdots A_j$ que separa o produto entre A_k e A_{k+1} contém soluções ótimas para os problemas de parentização de $A_i A_{i+1} \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$.

Você perceberá que está seguindo um padrão comum para descobrir a subestrutura ótima:

1. Mostrar que uma solução para o problema consiste em fazer uma escolha, como a de escolher um corte inicial em uma haste ou um índice no qual separar a cadeia de matrizes. Essa escolha produz um ou mais subproblemas a resolver.
2. Supor que, para um dado problema, existe uma escolha que resulta em uma solução ótima. Você ainda não se preocupa com a maneira de determinar essa escolha. Basta supor que ela existe.
3. Dada essa escolha, determinar quais subproblemas dela decorrem e como caracterizar melhor o espaço de subproblemas resultante.
4. Mostrar que as soluções para os subproblemas usados dentro de uma solução ótima para o problema também devem ser ótimas utilizando uma técnica de "recortar e colar". Para tal, você supõe que alguma das soluções de subproblemas não é ótima e, então, deduz uma contradição. Em particular, "recortando" a solução não ótima para cada subproblema e "colando" a solução ótima, você mostra que pode conseguir uma solução melhor para o problema original, o que contradiz a suposição de que você já tinha uma solução ótima. Se uma solução ótima der origem a mais de um subproblema, normalmente eles serão tão semelhantes que você pode modificar o argumento "recortar e colar" usado para um deles e aplicá-lo aos outros com pouco esforço.

Para caracterizar o espaço de subproblemas, uma boa regra prática é tentar manter o espaço tão simples quanto possível e depois expandi-lo conforme necessário. Por exemplo, o espaço de subproblemas que consideramos para o problema do corte da haste continha os problemas de determinar o corte ótimo para uma haste de comprimento i para cada tamanho i. Esse espaço de subproblemas funcionou bem e não tivemos nenhuma necessidade de tentar um espaço de subproblemas mais geral.

Ao contrário, suponha que tivéssemos tentado restringir nosso espaço de subproblemas para a multiplicação de cadeias de matrizes a produtos de matrizes da forma $A_1 A_2 \cdots A_J$. Como antes, uma parentização ótima deve separar esse produto entre A_k e A_{k+1} para algum $1 \le k \le j$. A menos que possamos garantir que k é sempre igual a $j - 1$, constataremos que tínhamos subproblemas da forma $A_1 A_2 \cdots A_k$ e $A_{k+1} A_{k+2} \cdots Aj$, e que este último subproblema não é da forma $A_1 A_2 \cdots A_j$. Para esse problema, tivemos que permitir que nossos subproblemas variassem em "ambas as extremidades", isto é, permitir que i e j variassem no subproblema $A_i A_{i+1} \cdots A_j$.

A subestrutura ótima varia nos domínios de problemas de duas maneiras:

1. o número de subproblemas usados em uma solução ótima para o problema original.
2. o número de opções que temos para determinar qual(is) subproblema(s) usar em uma solução ótima.

No problema do corte da haste, uma solução ótima para cortar uma haste de tamanho n usa apenas um subproblema (de tamanho $n – i$), mas temos de considerar n escolhas para i para determinar qual deles produz uma solução ótima. A multiplicação de cadeias de matrizes para a subcadeia $A_i A_i+1 \cdots A_j$ serve como um exemplo com dois subproblemas e $j - i$ escolhas. Para uma dada matriz A_k na qual separamos o produto, temos dois subproblemas — a parentização de $A_i A_i+1 \cdots A_k$ e a parentização de $A_k+1\ A_k+2 \cdots A_j$ — e devemos resolver *ambos* otimamente. Uma vez determinadas as soluções ótimas para subproblemas, escolhemos entre $j – i$ candidatos para o índice k.

Informalmente, o tempo de execução de um algoritmo de programação dinâmica depende do produto de dois fatores: o número global de subproblemas e quantas escolhas consideramos que existem para cada subproblema. No corte de hastes tínhamos $Q(n)$ subproblemas no total e no máximo n escolhas para examinar para cada um, resultando no tempo de execução $O(n_2)$. Na multiplicação de cadeias de matrizes, tínhamos $Q(n_2)$ subproblemas no total e, em cada um deles, tínhamos no máximo $n – 1$ escolhas, dando um tempo de execução $O(n_3)$ (na verdade, um tempo de execução $Q(n_3)$ pelo Exercício 15.2-5).

Normalmente, o grafo de subproblemas dá um modo alternativo de executar a mesma análise. Cada vértice corresponde a um subproblema, e as escolhas para um subproblema são as arestas que nele incidem. Lembre-se de que, no corte de hastes, o grafo do subproblema tinha n vértices e no máximo n arestas por vértice, resultando no tempo de execução $O(n_2)$. Na multiplicação de cadeia de matrizes, o grafo de subproblemas, se o tivéssemos desenhado, teria $Q(n_2)$ vértices e cada vértice teria um grau de no máximo $n – 1$, o que resultaria em um total de $O(n_3)$ vértices e arestas.

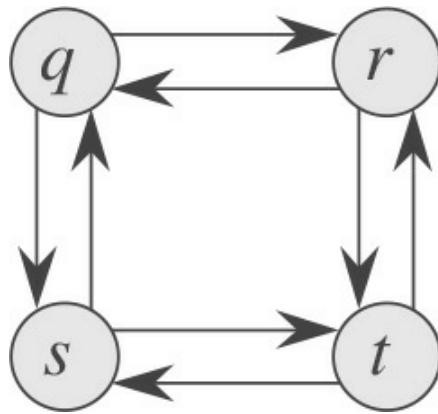
A programação dinâmica usa frequentemente a subestrutura ótima de baixo para cima. Isto é, primeiro encontramos soluções ótimas para subproblemas e, resolvidos os subproblemas, encontramos uma solução ótima para o problema. Encontrar uma solução ótima para o problema acarreta escolher, entre os subproblemas, aqueles que usaremos na solução do problema. Normalmente, o custo da solução do problema é igual aos custos dos subproblemas, mais um custo atribuível diretamente à escolha em si. Por exemplo, no corte de hastes, primeiro resolvemos os subproblemas de determinar maneiras ótimas de cortar hastes de comprimento i para $i = 0, 1,...,n – 1$ e depois determinamos qual subproblema produz uma solução ótima para uma haste de comprimento n, usando a equação (15.2). O custo atribuível à escolha em si é o termo p_i na equação (15.2). Na multiplicação de cadeias de matrizes, determinamos a parentização ótima de subcadeias de $A_i A_i+1 \cdots A_j$, e então escolhemos a matriz A_k na qual separar o produto. O custo atribuível à escolha propriamente dita é o termo $p_i - 1\ p_k p_j$.

No Capítulo 16, examinaremos os “algoritmos gulosos”, que guardam muitas semelhanças com a programação dinâmica. Em particular, os problemas aos quais os algoritmos gulosos se aplicam têm subestrutura ótima. Uma diferença notável entre algoritmos gulosos e programação dinâmica é que, em vez de primeiro encontrar soluções ótimas para subproblemas e depois fazer uma escolha informada, os algoritmos gulosos primeiro fazem uma escolha “gulosa” — a que parece melhor no momento — e depois resolvem um problema resultante, sem se darem ao trabalho de resolver todos os possíveis subproblemas menores relacionados.

Surpreendentemente, em alguns casos a estratégia funciona!

Sutilezas

Devemos ter cuidado para não presumir que a subestrutura ótima seja aplicável quando não é. Considere os dois problemas a seguir, nos quais temos um grafo dirigido $G = (V, E)$ e vértices $u, v \in V$.

Caminho mais curto não ponderado:[3] Encontrar um caminho de u para v que consista no menor número de arestas. Tal caminho deve ser simples, já que remover um ciclo de um caminho produz um caminho com menos arestas.

Caminho simples mais longo não ponderado: Encontrar um caminho simples de u para v que consista no maior número de arestas. Precisamos incluir o requisito de simplicidade porque, do contrário, acabamos percorrendo um ciclo tantas vezes quantas quisermos para criar caminhos com um número arbitrariamente grande de arestas.

O problema do caminho mais curto não ponderado exibe subestrutura ótima, como mostramos a seguir. Suponha que $u \neq v$, de modo que o problema é não trivial. Então, qualquer caminho p de u para v deve conter um vértice intermediário, digamos w. (Observe que w pode ser u ou v.) Assim, podemos decompor o caminho $u\ p\ v$ em subcaminhos $u\ p_1\ w\ p_2\ v$. É claro que o número de arestas em p é igual ao número de arestas em p_1 mais o número de arestas em p_2. Afirmamos que, se p é um caminho ótimo (isto é, o mais curto) de u para v, então p_1 deve ser um caminho mais curto de u para w. Por quê? Usamos um argumento de “recortar e colar”: se existisse outro caminho, digamos de u para w com menos arestas que p'_1, poderíamos recortar p_1 e colar em p'_1 para produzir um caminho $u\ p'_1\ w\ p_2\ v$ com menos arestas que p, assim contra-dizendo a otimalidade de p. Simetricamente, p_2 deve ser um caminho mais curto de w para v. Assim, podemos encontrar um caminho mais curto de u para v considerando todos os vértices intermediários w, encontrando um caminho mais curto de u para w e um caminho mais curto de w para v, e escolhendo um vértice intermediário w que produza o caminho mais curto global. Na Seção 25.2, usamos uma variante dessa observação de subestrutura ótima para encontrar um caminho mais curto entre cada par de vértices em um grafo ponderado e dirigido.

É tentador supor que o problema de encontrar um caminho simples mais longo não ponderado também exibe subestrutura ótima. Afinal, se desdobrarmos um caminho simples mais longo $u \overset{p}{\leadsto} v$ em subcaminhos $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$, então p não deve ser um caminho simples mais longo de u para w, e p_2 não deve ser um caminho simples mais longo de w para v? A resposta é não! A Figura 15.6 nos dá um exemplo. Considere o caminho $q \to r \to t$, que é um caminho simples mais longo de q para t. O caminho $q \to r$ é um caminho simples mais longo de q para r? Não, já que o caminho $q \to s \to t \to r$ é um caminho simples mais longo. $r \to t$ é um caminho simples mais longo de r para t? Novamente não, já que o caminho $r \to q \to s \to t$ é um caminho simples mais longo.

Esse exemplo mostra que, para caminhos simples mais longos, não apenas falta uma subestrutura ótima para o problema, mas tampouco podemos montar necessariamente uma solução “legítima” para o problema a partir de soluções para subproblemas. Se combinarmos os caminhos simples mais longos $q \to s \to t \to r$ e $r \to q \to s \to t$, obteremos o caminho $q \to s \to t \to r \to q \to s \to t$, que não é simples. Na realidade, o problema de encontrar um caminho simples mais longo não ponderado não parece ter nenhum tipo de subestrutura ótima. Nenhum algoritmo eficiente de programação dinâmica foi encontrado para esse problema até hoje. De fato, esse problema é NP-completo, que — como veremos no Capítulo 34 — significa que é improvável que ele possa ser resolvido em tempo polinomial.

Por que a subestrutura de um caminho simples mais longo é tão diferente da subestrutura de um caminho mais curto? Embora uma solução para um problema para ambos, o caminho mais longo e o mais curto, use dois subproblemas, os subproblemas para encontrar o caminho simples mais longo não são ***independentes***, mas o são para caminhos mais curtos. O que significa subproblemas independentes? Significa que a solução para um subproblema não afeta a solução para outro subproblema do mesmo problema. No exemplo da Figura 15.6, temos o problema de encontrar um caminho simples mais longo de q para t com dois subproblemas: encontrar caminhos simples mais longos de q para r e de r para t. Para o primeiro desses subproblemas, escolhemos o caminho $q \to s \to t \to r$ e, portanto, também usamos os vértices s e t. Não podemos mais usar esses vértices no segundo subproblema, já que a combinação das duas soluções para subproblemas produziria um caminho que não é simples. Se não podemos usar o vértice t no segundo problema, não podemos resolvê-lo, já que t tem de estar no caminho que encontrarmos, e ele não é o vértice no qual estamos “encaixando” as soluções do subproblema (esse vértice é r). Como usamos os vértices s e t em uma solução de subproblema, não podemos usá-los na solução do outro subproblema. Porém, temos de usar, no mínimo, um deles para resolver o outro subproblema e temos de usar ambos para resolvê-lo otimamente. Assim, dizemos que esses subproblemas não são independentes. Visto de outro modo, usar recursos para resolver um subproblema (sendo esses recursos os vértices) torna-os indisponíveis para o outro subproblema.

Figura 15.6 Um grafo dirigido mostrando que o problema de encontrar um caminho simples mais longo em um grafo dirigido não ponderado não tem subestrutura ótima. O caminho $q \to r \to t$ é um caminho simples mais longo de q para t, mas o subcaminho $q \to r$ não é um caminho simples mais longo de q para r nem o subcaminho $r \to t$ é um caminho simples mais longo de r para t.

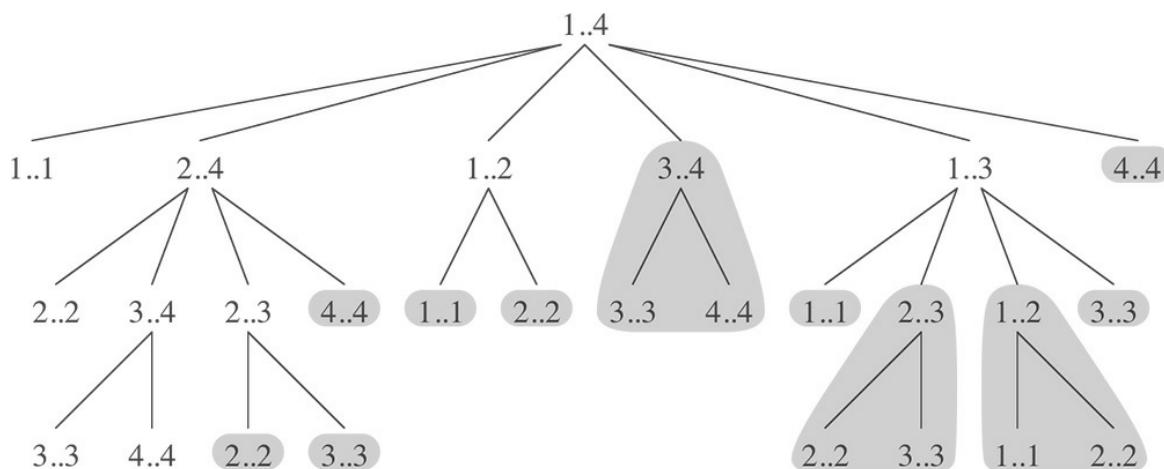
Então, por que os subproblemas são independentes para encontrar um caminho mais curto? A resposta é que, por natureza, os subproblemas não compartilham recursos. Afirmamos que, se um vértice w está em um caminho mais curto p de u para v, então podemos emendar *qualquer* caminho mais curto $u \overset{p_1}{\leadsto} w$ e *qualquer* caminho mais curto $w \overset{p_2}{\leadsto} v$ para produzir um caminho mais curto de u para v. Estamos seguros de que, além de w, nenhum vértice pode aparecer nos caminhos p_1 e p_2. Por quê? Suponha que algum vértice $x \neq w$ apareça tanto em p_1 quanto em p_2, de modo que podemos decompor p_1 como $u \overset{p_{ux}}{\leadsto} x \sim w$ e p_2 como $w \overset{p_{xv}}{\leadsto} v$. Pela subestrutura ótima desse problema, o caminho p tem tantas arestas quanto p_1 e p_2 juntos; vamos dizer que p tenha e arestas. Agora, vamos construir um caminho $p' = u\ p_{ux}$ $w\ p_{xv}\ v$ de u para v. Como cortamos os caminhos de x para w e de w para x e cada um deles contém no mínimo uma aresta, o caminho p_0 contém no máximo $e - 2$ arestas, o que contradiz a hipótese de p ser um caminho mais curto. Assim, estamos seguros de que os subproblemas para o problema do caminho mais curto são independentes.

Ambos os problemas examinados nas Seções 15.1 e 15.2 têm subproblemas independentes. Na multiplicação de cadeias de matrizes, os subproblemas são multiplicar subcadeias $A_i A_i{+1} \cdots A_k$ e $A_k{+1}\ A_k{+2} \cdots A_j$. Essas subcadeias são disjuntas, de modo que não haveria possibilidade de alguma matriz ser incluída em ambas. No corte de hastes, para determinar o melhor modo de cortar uma haste de comprimento n, consideramos os melhores modos de cortar hastes de comprimento i para $i = 0, 1, ..., n - 1$. Como uma solução ótima para o problema do comprimento n inclui apenas uma dessas soluções de subproblemas (após termos cortado o primeiro pedaço), a independência de subproblemas nem entra no assunto.

Subproblemas sobrepostos

O segundo elemento que um problema de otimização deve ter para a programação dinâmica ser aplicável é que o espaço de subproblemas deve ser "pequeno", no sentido de que um algoritmo recursivo para o problema resolve os mesmos subproblemas repetidas vezes, em lugar de sempre gerar novos subproblemas. Em geral, o número total de subproblemas distintos é um polinômio no tamanho de entrada. Quando um algoritmo recursivo reexamina o mesmo problema repetidamente, dizemos que o problema de otimização tem ***subproblemas sobrepostos***.[4] Ao contrário, um problema para o qual uma abordagem de divisão e conquista é adequada, normalmente gera problemas absolutamente novos em cada etapa da recursão. Algoritmos de programação dinâmica, normalmente tiram proveito de subproblemas sobrepostos resolvendo cada subproblema uma vez e depois armazenando a solução em uma tabela onde ela pode ser examinada quando necessário, usando um tempo constante por busca.

Na Seção 15.1, examinamos brevemente como uma solução recursiva para o problema do corte de hastes faz exponencialmente muitas chamadas para encontrar soluções para problemas menores. Nossa solução de programação dinâmica toma um algoritmo recursivo de tempo exponencial e o reduz a um algoritmo de tempo linear.

Para ilustrar a propriedade de subproblemas sobrepostos com mais detalhes, vamos examinar novamente o problema de multiplicação de cadeias de matrizes. Consultando mais uma vez a Figura 15.5, observe que MATRIX-CHAIN-ORDER consulta repetidamente a solução para subproblemas em linhas inferiores quando está resolvendo subproblemas em linhas superiores. Por exemplo, referencia a entrada $m[3, 4]$ quatro vezes: durante os cálculos de $m[2, 4]$, $m[1, 4]$, $m[3, 5]$ e $m[3, 6]$. Se tivéssemos de recalcular $m[3, 4]$ toda vez, em vez de apenas consultá-la, o tempo de execução aumentaria expressivamente. Para ver como, considere o seguinte procedimento recursivo (ineficiente) que determina $m[i, j]$, o número mínimo de multiplicações escalares necessárias para calcular o produto de cadeias de matrizes $A_{i..j} = A_i A_{i+1} \cdots A_j$. O procedimento é diretamente baseado na recorrência (15.7).

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

A Figura 15.7 mostra a árvore de recursão produzida pela chamada RECURSIVE-MATRIX--CHAIN(p, 1, 4). Cada nó é identificado pelos valores dos parâmetros i e j. Observe que alguns pares de valores ocorrem muitas vezes.

De fato, podemos mostrar que o tempo para calcular $m[1, n]$ por esse procedimento recursivo é no mínimo exponencial em n. Seja $T(n)$ o tempo tomado por RECURSIVE-MATRIX-CHAIN para calcular uma parentização ótima de uma cadeia de n matrizes. Como cada uma das execuções das linhas 1-2 e das linhas 6-7 demora no mínimo tempo unitário, assim como a multiplicação na linha 5, a inspeção do procedimento produz a recorrência

Figura 15.7 A árvore de recursão para a execução de RECURSIVE-MATRIX-CHAIN(p, 1, 4). Cada nó contém os parâmetros i e j. Os cálculos executados em uma subárvore sombreada são substituídos por uma única consulta à tabela em MEMOIZED-MATRIX-CHAIN.

$$T(1) \geq 1,$$
$$T(n) \geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1) \quad \text{para } n > 1.$$

Observando que, para $i = 1, 2, ..., n - 1$, cada termo $T(i)$ aparece uma vez como $T(k)$ e uma vez como $T(n - k)$, e reunindo os $n - 1$ valores 1 no somatório com o valor 1 à frente, podemos

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n. \tag{15.8}$$

Provaremos que $T(n) = \Omega(2_n)$ usando o método de substituição. Especificamente, mostraremos que $T(n) \geq 2_n$ - 1 para todo $n \geq 1$. A base é fácil, já que $T(1) \geq 1 = 2^0$. Por indução, para $n \geq 2$, temos

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \quad \text{(pela equação (A.5))} \\ &= 2^n - 2 + n \\ &\geq 2^{n-1}, \end{aligned}$$

o que conclui a prova. Assim, a quantidade total de trabalho executado pela chamada RECURSIVE-MATRIX-CHAIN$(p, 1, n)$ é no mínimo exponencial em n.

Compare esse algoritmo recursivo de cima para baixo (sem memoização) com o algoritmo de programação dinâmica de baixo para cima. Este último é mais eficiente porque tira proveito da propriedade de subproblemas sobrepostos. A multiplicação de cadeias de matrizes tem somente Q(n_2) subproblemas distintos, e o algoritmo de programação dinâmica resolve cada um deles exatamente uma vez. Por outro lado, novamente o algoritmo recursivo deve resolver cada subproblema toda vez que ele reaparece na árvore de recursão. Sempre que uma árvore de recursão para a solução recursiva natural para um problema contiver o mesmo subproblema repetidamente e o número total de subproblemas distintos for pequeno, a programação dinâmica pode melhorar a eficiência, às vezes, expressivamente.

Reconstrução de uma solução ótima

Como regra prática, muitas vezes, armazenamos em uma tabela a opção que escolhemos em cada subproblema, de modo que não tenhamos de reconstruir essa informação com base nos custos que armazenamos.

Na multiplicação de cadeias de matrizes, a tabela $s[i, j]$ poupa uma quantidade significativa de trabalho durante a reconstrução de uma solução ótima. Suponha que não mantivéssemos a tabela $s[i, j]$, tendo preenchido apenas a tabela $m[i, j]$, que contém custos de subproblemas ótimos. Escolhemos entre $j - i$ possibilidades quando determinamos quais subproblemas usar em uma solução ótima para parentizar $A_i\, A_i$+1 $\cdots$ A_j, e $j - i$ não é uma constante. Portanto, demoraria o tempo Q$(j - i) = \omega(1)$ para reconstruir os subproblemas que escolhemos para uma solução de um problema dado. Armazenando em $s[i, j]$ o índice da matriz nos quais separamos o produto $A_i\, A_i$+1 $\cdots$ A_j, podemos reconstruir cada escolha no tempo $O(1)$.

Memoização

Como vimos no problema do corte de hastes, há uma abordagem alternativa para a programação dinâmica que frequentemente oferece a eficiência da abordagem de programação dinâmica de baixo para cima e ao mesmo tempo mantém uma estratégia de cima para baixo. A ideia é ***memoizar*** o algoritmo recursivo natural mas ineficiente. Como na abordagem de baixo para cima, mantemos uma tabela com soluções de subproblemas, mas a estrutura de controle para preencher a tabela é mais semelhante ao algoritmo recursivo.

Um algoritmo recursivo memoizado mantém uma entrada em uma tabela para a solução de cada subproblema. Cada entrada da tabela contém inicialmente um valor especial para indicar que a entrada ainda tem de ser preenchida. Quando o subproblema é encontrado pela primeira vez durante a execução do algoritmo recursivo, sua solução é calculada e depois armazenada na tabela. Cada vez subsequente que encontrarmos esse subproblema, simplesmente consultamos o valor armazenado na tabela e o devolvemos.[5]

Apresentamos a seguir, uma versão memoizada de RECURSIVE-MATRIX-CHAIN. Observe os pontos de semelhança com o método de cima para baixo memoizado para o problema do corte de hastes.

```
MEMOIZED-MATRIX-CHAIN (p)

1  n = p.comprimento − 1
2  seja m[1 .. n, 1 .. n] uma nova tabela
3  for i = 1 to n
4     for j = i to n
5         m[i, j] = ∞
6  return LOOKUP-CHAIN (m, p, 1, n)
```

```
LOOKUP-CHAIN (m, p, i, j)

1  if m[i, j] < ∞
2     return m[i, j]
3  if i == j
4     m[i, j] = 0
5  else for k = i to j − 1
6           q = LOOKUP-CHAIN (m, p, i, k)
                + LOOKUP-CHAIN (m, p, k + 1, j) + p_{i − 1} p_k p_j
7           if q < m[i, j]
8               m[i, j] = q
9  return m[i, j]
```

O procedimento MEMOIZED-MATRIX-CHAIN, assim como o procedimento MATRIX-CHAIN-ORDER, mantém uma tabela $m[1..n, 1..n]$ de valores calculados de $m[i, j]$, o número mínimo de multiplicações escalares necessárias para calcular a matriz $A_{i..j}$. Cada entrada de tabela contém inicialmente o valor ∞ para indicar que a entrada ainda tem de ser preenchida. Quando a chamada LOOKUP-CHAIN(p, i, j) é executada, se a linha 1 verificar que $m[i, j] < \infty$, o procedimento simplesmente retorna o custo $m[i, j]$ calculado anteriormente na linha 2. Caso contrário, o custo é calculado como em RECURSIVE-MATRIX-CHAIN, armazenado em $m[i, j]$ e retornado. Assim, LOOKUP-CHAIN(p, i, j) sempre retorna o valor de $m[i, j]$, mas só o calcula na primeira chamada de LOOKUP-CHAIN que tenha esses valores específicos de i e j.

A Figura 15.7 ilustra como MEMOIZED-MATRIX-CHAIN poupa tempo em comparação com RECURSIVE-MATRIX-CHAIN. As subárvores sombreadas representam valores que o procedimento consulta em vez de recalcular.

Do mesmo modo que o algoritmo de programação dinâmica de baixo para cima MATRIX-CHAIN-ORDER, o procedimento MEMOIZED-MATRIX-CHAIN é executado em tempo $O(n_3)$. A linha 5 de MEMOIZED-MATRIX-CHAIN é executada $O(n_2)$ vezes. Podemos classificar as chamadas de LOOKUP-CHAIN em dois tipos:

1. Chamadas nas quais $mi, j = \infty$, de modo que as linhas 3-9 são executadas.
2. Chamadas nas quais $mi, j < \infty$, de modo que LOOKUP-CHAIN simplesmente retorna na linha 2.

Há Q(n_2) chamadas do primeiro tipo, uma por entrada de tabela. Todas as chamadas do segundo tipo são feitas como chamadas recursivas por chamadas do primeiro tipo. Sempre que uma dada chamada de LOOKUP-CHAIN faz chamadas recursivas, ela faz $O(n)$ chamadas. Assim, há ao todo $O(n_3)$ chamadas do segundo tipo. Cada chamada do segundo tipo demora o tempo $O(1)$, e cada chamada do primeiro tipo demora o tempo $O(n)$ mais o tempo gasto em suas chamadas recursivas. Portanto, o tempo total é $O(n_3)$. Assim, a memoização transforma um algoritmo de tempo (2_n) em um algoritmo de tempo $O(n_3)$.

Resumindo, podemos resolver o problema de multiplicação de cadeias de matrizes no tempo $O(n_3)$ por um algoritmo de programação dinâmica de cima para baixo memoizado ou por um algoritmo de programação dinâmica de baixo para cima. Ambos os métodos tiram proveito da propriedade dos subproblemas sobrepostos. Há apenas Q(n_2) subproblemas distintos no total e qualquer um desses métodos calcula a solução para cada subproblema somente uma vez. Sem memoização, o algoritmo recursivo natural é executado em tempo exponencial, já que subproblemas resolvidos são resolvidos repetidas vezes.

Na prática geral, se todos os subproblemas devem ser resolvidos no mínimo uma vez, o desempenho de um algoritmo de programação dinâmica de baixo para cima, normalmente supera o de um algoritmo de cima para baixo memoizado por um fator constante porque o algoritmo de baixo para cima não tem nenhuma sobrecarga para recursão e a sobrecarga associada à manutenção da tabela é menor. Além disso, em alguns problemas podemos explorar o padrão regular de acessos a tabelas no algoritmo de programação dinâmica para reduzir ainda mais os requisitos de tempo ou espaço. Alternativamente, se alguns subproblemas no espaço de subproblemas não precisarem ser resolvidos de modo algum, a solução memoizada tem a vantagem de resolver somente os subproblemas que são definitivamente necessários.

Exercícios

15.3-1 Qual modo é mais eficiente para determinar o número ótimo de multiplicações em um problema de multiplicação de cadeias de matrizes: enumerar todos os modos de parentizar o produto e calcular o número de multiplicações para cada um ou executar RECURSIVE-MATRIX-CHAIN? Justifique sua resposta.

15.3-2 Desenhe a árvore de recursão para o procedimento MERGE-SORT da Seção 2.3.1 em um arranjo de 16 elementos. Explique por que a memoização não aumenta a velocidade de um bom algoritmo de divisão e conquista como MERGE-SORT.

15.3-3 Considere uma variante do problema da multiplicação de cadeias de matrizes na qual a meta é parentizar a sequência de matrizes de modo a maximizar, em vez de minimizar, o número de multiplicações escalares. Esse problema exibe subestrutura ótima?

15.3-4 Como já dissemos, em programação dinâmica primeiro resolvemos os subproblemas e depois escolhemos qual deles utilizar em uma solução ótima para o problema. A professora Capulet afirma que nem sempre é necessário resolver todos os subproblemas para encontrar uma solução ótima. Ela sugere que podemos encontrar uma solução ótima para o problema de multiplicação de cadeias de matrizes escolhendo sempre a matriz A_k na qual separar o subproduto $A_i A_i{+1} \cdots A_j$ (selecionando k para minimizar a quantidade $p_i - {}^1 p_k p_j$) *antes* de resolver os subproblemas. Determine uma instância do problema de multiplicação de cadeias de matrizes para a qual essa abordagem gulosa produz uma solução subótima.

15.3-5 Suponha que, no problema do corte de hastes da Seção 15.1, tivéssemos também o limite l_i para o número de peças de comprimento i que era permitido produzir, para i = 1, 2,...,n. Mostre que a propriedade de subestrutura ótima descrita na Seção 15.1 deixa de ser válida.

15.3-6 Imagine que você queira fazer o câmbio de uma moeda por outra e percebe que, em vez de trocar diretamente uma moeda por outra, seria melhor efetuar uma série de trocas intermediárias por outras moedas, por fim tendo em mãos a moeda que queria. Suponha que você possa trocar n moedas diferentes, numeradas de $1,2,...,n$, que começará com a moeda 1 e quer terminar com a moeda n. Você tem, para cada par de moedas i e j, uma taxa de câmbio r_{ij} , o que significa que, se você começar com d unidades da moeda i, poderá trocá-las por dr_{ij} unidades da moeda j. Uma sequência de trocas pode acarretar uma comissão, que depende do número de trocas que você faz. Seja c_k a comissão cobrada quando você faz k trocas. Mostre que, se $c_k = 0$ para todo $k = 1,2,...,n$, o problema de determinar a melhor sequência de trocas da moeda 1 para a moeda n exibe subestrutura ótima. Então, mostre que, se as comissões c_k são valores arbitrários, o problema de determinar a melhor sequência de trocas da moeda 1 para a moeda n não exibe necessariamente subestrutura ótima.

15.4 SUBSEQUÊNCIA COMUM MAIS LONGA

Em aplicações biológicas, muitas vezes, é preciso comparar o DNA de dois (ou mais) organismos diferentes. Um filamento de DNA consiste em uma cadeia de moléculas denominadas ***bases***, na qual as bases possíveis são adenina, guanina, citosina e timina. Representando cada uma dessas bases por sua letra inicial, podemos expressar um filamento de DNA como uma cadeia no conjunto finito {A,C,G,T}. (O Apêndice C dá a definição de uma cadeia.) Por exemplo, o DNA de um organismo pode ser S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, e o DNA de outro organismo pode ser S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA. Uma razão para a comparação de dois filamentos de DNA é determinar o grau de "semelhança" entre eles, que serve como alguma medida da magnitude da relação entre os dois organismos. Podemos definir (e definimos) a semelhança de muitas maneiras diferentes. Por exemplo, podemos dizer que dois filamentos de DNA são semelhantes se um deles for uma subcadeia do outro. (O Capítulo 32 explora algoritmos para resolver esse problema.) Em nosso exemplo, nem S_1 nem S_2 é uma subcadeia do outro. Alternativamente, poderíamos dizer que dois filamentos são semelhantes se o número de mudanças necessárias para transformar um no outro for pequeno. (O Problema 15-3 explora essa noção.) Ainda uma outra maneira de medir a semelhança entre filamentos S_1 e S_2 é encontrar um terceiro filamento S_3 no qual as bases em S_3 aparecem em cada um dos filamentos S_1 e S_2; essas bases devem aparecer na mesma ordem, mas não precisam ser necessariamente consecutivas. Quanto mais longo o filamento S_3 que pudermos encontrar, maior será a semelhança entre S_1 e S_2 . Em nosso exemplo, o filamento S_3 mais longo é GTCGTCGGAAGCCGGCCGAA.

Formalizamos essa última noção de semelhança como o problema da subsequência comum mais longa. Uma subsequência de uma determinada sequência é apenas a sequência dada na qual foram omitidos zero ou mais elementos. Em termos formais, dada uma sequência $X = \langle x_1, x_2, ..., x_m\rangle$, uma outra sequência $Z = \langle z_1, z_2, ..., z_k\rangle$ é uma ***subsequência*** de X se existir uma sequência estritamente crescente $\langle i_1, i_2, ..., i_k\rangle$ de índices de X tais que, para todo $j = 1, 2, ..., k$, temos $x_i = z_j$. Por exemplo, $Z = \langle B, C, D, B\rangle$ é uma subsequência de $X = \langle A, B, C, B, D, A, B\rangle$ com sequência de índices correspondente $\langle 2, 3, 5, 7\rangle$.

Dadas duas sequências X e Y, dizemos que uma sequência Z é uma ***subsequência comum*** de X e Y se Z é uma subsequência de X e Y. Por exemplo, se $X = \langle A, B, C, B, D, A, B\rangle$ e $Y = \langle B, D, C, A, B, A\rangle$, a sequência $\langle B, C, A\rangle$ é uma subsequência comum das sequências X e Y. Porém, a sequência $\langle B, C, A\rangle$ não é uma subsequência comum *mais longa* (LCS — longest common subsequence) de X e Y, já que tem comprimento 3, e a sequência $\langle B, C, B, A\rangle$, que também é comum a X e Y, tem comprimento 4. A sequência $\langle B, C, B, A\rangle$ é uma LCS de X e Y, assim como a sequência $\langle B, D, A, B\rangle$, visto que não existe nenhuma subsequência comum de comprimento 5 ou maior.

No ***problema da subsequência comum mais longa***, temos duas sequências $X = \langle x_1, x_2, ..., x_m\rangle$ e $Y = \langle y_1, y_2, ..., y_n\rangle$, e desejamos encontrar uma subsequência comum de comprimento máximo de X e Y. Esta seção mostra como resolver o problema da LCS eficientemente, usando programação dinâmica.

Etapa 1: Caracterização de uma subsequência comum mais longa

Uma abordagem de força bruta para resolver o problema da LCS seria enumerar todas as subsequências de X e conferir cada subsequência para ver se ela também é uma subsequência de Y, sem perder de vista a subsequência mais longa encontrada. Cada subsequência de X corresponde a um subconjunto dos índices $\{1, 2, ..., m\}$ de X. Como X tem 2^m subsequências, essa abordagem requer tempo exponencial, o que a torna impraticável para sequências longas.

Porém, o problema da LCS tem uma propriedade de subestrutura ótima, como mostra o teorema a seguir. Como veremos, as classes naturais de subproblemas correspondem a pares de "prefixos" das duas sequências de entrada. Mais precisamente, dada uma sequência $X = \langle x_1, x_2, .., x_m \rangle$, definimos o i-ésimo ***prefixo*** de X, para $i = 0, 1, ..., m$, como $X_i = \langle x_1, x_2, ..., x_i \rangle$. Por exemplo, se $X = \langle A, B, C, B, D, A, B \rangle$, então $X_4 = \langle A, B, C, B \rangle$ e X_0 é a sequência vazia.

Teorema 15.1 (Subestrutura ótima de uma LCS)

Sejam $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$ as sequências, e seja $Z = \langle z_1, z_2, ..., z_k \rangle$ qualquer LCS de X e Y.

1. Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} é uma LCS de X_{m-1} e Y_{n-1}.
2. Se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma LCS de X_{m-1} e Y.
3. Se $x_m \neq y_n$, então $z_k \neq y_m$ implica que Z é uma LCS de X e Y_{n-1}.

Prova (1) Se $z_k \neq x_m$, então podemos anexar $x_m = y_n$ a Z para obter uma subsequência comum de X e Y de comprimento $k + 1$, contradizendo a suposição de que Z é uma subsequência comum *mais longa* de X e Y. Assim, devemos ter $z_k = x_m = y_n$. Agora, o prefixo Z_{k-1} é uma subsequência comum de comprimento $(k - 1)$ de X_{m-1} e Y_{n-1}. Desejamos mostrar que ela é uma LCS. Suponha, por contradição, que exista uma subsequência comum W de X_{m-1} e Y_{n-1} com comprimento maior que $k - 1$. Então, anexar $x_m = y_n$ a W produz uma subsequência comum de X e Y cujo comprimento é maior que k, o que é uma contradição.

(2) Se $z_k \neq x_m$, então Z é uma subsequência comum de X_{m-1} e Y. Se existisse uma subsequência comum W de X_{m-1} e Y com comprimento maior que k, então W seria também uma subsequência comum de X_m e Y, contradizendo a suposição de que Z é uma LCS de X e Y.

(3) A prova é simétrica a (2).

O modo como o Teorema 15.1 caracteriza subsequências comuns mais longas nos diz que uma LCS de duas sequências contém uma LCS de prefixos das duas sequências. Assim, o problema de LCS tem uma propriedade de subestrutura ótima. Uma solução recursiva também tem a propriedade de subproblemas sobrepostos, como veremos em breve.

Etapa 2: Uma solução recursiva

O Teorema 15.1 subentende que devemos examinar um ou dois subproblemas quando queremos encontrar uma LCS de $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$. Se $x_m = y_n$, devemos encontrar uma LCS de X_{m-1} e Y_{n-1}. Anexar $x_m = y_n$ a essa LCS produz uma LCS de X e Y. Se $x_m \neq y_n$, então devemos resolver dois subproblemas: encontrar uma LCS de X_{m-1} e Y e encontrar uma LCS de X e Y_{n-1}. A mais longa de qualquer dessas duas LCS é uma LCS de X e Y. Como esses casos esgotam todas as possibilidades, sabemos que uma das soluções ótimas de subproblemas certamente aparecerá dentro de uma LCS de X e Y.

É fácil ver a propriedade de subproblemas sobrepostos no problema da LCS. Para encontrar uma LCS de X e Y, pode ser necessário encontrar as LCS de X e Y_{n-1} e de X_{m-1} e Y. Porém, cada um desses subproblemas tem o subsubproblema de encontrar uma LCS de X_{m-1} e Y_{n-1}. Muitos outros subproblemas compartilham subsubproblemas.

Como ocorreu no problema de multiplicação de cadeias de matrizes, nossa solução recursiva para o problema da LCS envolve estabelecer uma recorrência para o valor de uma solução ótima. Vamos definir $c[i, j]$ como o

comprimento de uma LCS das sequências X_i e Y_j. Se $i = 0$ ou $j = 0$, uma das sequências tem comprimento 0 e, portanto, a LCS tem comprimento 0. A subestrutura ótima do problema da LCS dá a fórmula recursiva

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_i, \\ \max(c[i, j-1],\, c[i-1,\, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j. \end{cases} \quad (15.9)$$

Observe que, nessa formulação recursiva, uma condição no problema restringe os subproblemas que podemos considerar. Quando $x_i = y_j$, podemos e devemos considerar o subproblema de encontrar a LCS de X_i-1 e Y_j-1. Caso contrário, consideramos os dois subproblemas de encontrar uma LCS de X_i e Y_j-1 e de X_i-1 e Y_j. Nos algoritmos de programação dinâmica que já examinamos — para corte de hastes e para multiplicação de cadeias de matrizes —, não descartamos nenhum subproblema por causa de condições no problema. O algoritmo para encontrar uma LCS não é o único algoritmo de programação dinâmica que descarta subproblemas com base em condições no problema. Por exemplo, o problema da distância de edição (ver o Problema 15-3) tem essa característica.

Etapa 3: Cálculo do comprimento de uma LCS

Tendo como base a equação (15.9), seria fácil escrever um algoritmo recursivo de tempo exponencial para calcular o comprimento de uma LCS de duas sequências. Contudo, visto que o problema da LCS tem somente Q(mn) subproblemas distintos, podemos usar programação dinâmica para calcular as soluções de baixo para cima.

O procedimento LCS-LENGTH toma duas sequências $X = \langle x_1, x_2, ..., x_m\rangle$ e $Y = \langle y_1, y_2, ..., y_n\rangle$ como entradas. Armazena os valores $c[i, j]$ em uma tabela $c[0 .. m, 0 .. n]$ e calcula as entradas em ***ordem orientada por linha*** (isto é, preenche a primeira linha de c da esquerda para a direita, depois a segunda linha, e assim por diante). O procedimento também mantém a tabela $b[1 .. m, 1 .. n]$ para ajudar a construir uma solução ótima. Intuitivamente, $b[i, j]$ aponta para a entrada da tabela correspondente à solução ótima de subproblema escolhida ao se calcular $c[i, j]$. O procedimento retorna as tabelas b e c; $c[m, n]$ contém o comprimento de uma LCS de X e Y.

```
LCS-LENGTH (X, Y)

1   m = X.comprimento
2   n = Y.comprimento
3   sejam b[1..m, 1..n] e c[0..m, 0..n] tabelas novas
4   for i = 1 to m
5      c[i,0] = 0
6   for j = 0 to n
7      c[0, j] = 0
8   for i = 1 to m
9      for j = 1 to n
10         if x_i == y_j
11             c[i, j] = c[i − 1, j − 1] + 1
12             b[i, j] = "↖"
13         elseif c[i − 1, j] ≥ c[i, j − 1]
14             c[i, j] = c[i − 1, j]
15             b[i, j] = "↑"
16         else c[i, j] = c[i, j- 1]
17             b[i, j] = "←"
18  return c, b
```

A Figura 15.8 mostra as tabelas produzidas por LCS-LENGTH nas sequências $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$. O tempo de execução do procedimento é Q(mn), já que cada entrada de tabela demora o tempo Q(1) para ser calculada.

Etapa 4: Construção de uma LCS

A tabela b retornada por LCS-LENGTH nos habilita a construir rapidamente uma LCS de $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$. Simplesmente começamos em $b[m, n]$ e percorremos a tabela seguindo as setas. Sempre que encontramos uma “ ” na entrada $b[i, j]$, ela implica que $x_i = y_j$ é um elemento da LCS que LCS-LENGTH encontrou. Com esse método, encontramos os elementos da LCS em ordem inversa. O procedimento recursivo a seguir imprime uma LCS de X e Y na ordem direta adequada. A invocação inicial é PRINT-LCS(b, X, $X.comprimento$, $Y.comprimento$).

```
PRINT-LCS(b, X, i, j)
1  if i == 0 ou j == 0
2     return
3  if b[i, j] == "↖"
4     PRINT-LCS(b, X, i − 1, j − 1)
5     print x_i
6  elseif b[i, j] == "↑"
7     PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Figura 15.8 As tabelas c e b calculadas por LCS-LENGTH para as sequências $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$. O quadrado na linha i e coluna j contém o valor de $c_{i,j}$ e a seta adequada para o valor de $b_{i,j}$. A entrada 4 em $c_{7,6}$ — o canto inferior direito da tabela — é o comprimento de uma LCS $\langle B, C, B, A \rangle$ de X e Y. Para $i, j > 0$, a entrada $c_{i,j}$ depende apenas de $x_i = y_j$ e dos valores nas entradas $c_{i-1,j}$, $c_{i,j-1}$ e $c_{i-1,j-1}$, que são calculados antes de $c_{i,j}$. Para reconstruir os elementos de uma LCS, siga as setas $b_{i,j}$ desde o canto inferior direito; a sequência está sombreada. Cada "\" na sequência sombreada corresponde a uma entrada (destacada) para a qual $x_i = y_j$ é membro de uma LCS.

Para a tabela b na Figura 15.8, esse procedimento imprime $BCBA$. O procedimento demora o tempo $O(m + n)$, já que decrementa no mínimo um de i e j em cada fase da recursão.

Melhorando o código

Depois de ter desenvolvido um algoritmo, você, muitas vezes, constatará que é possível melhorar o tempo ou o espaço que ele utiliza. Algumas mudanças podem simplificar o código e melhorar fatores constantes, porém, quanto ao

mais, não produzem nenhuma melhora assintótica no desempenho. Outras podem resultar em economias assintóticas significativas de tempo e de espaço.

Por exemplo, no algoritmo LCS podemos eliminar totalmente a tabela b. Cada entrada $c[i, j]$ depende apenas de três outras entradas na tabela $c[i-1, j-1]$, $c[i-1, j]$ e $c[i, j-1]$. Dado o valor de $c[i, j]$, podemos determinar em tempo $O(1)$ de qual desses três valores foi usado para calcular $c[i, j]$, sem inspecionar a tabela b. Assim, podemos reconstruir uma LCS em tempo $O(m + n)$ usando um procedimento semelhante a Print-LCS. (O Exercício 15.4-2 pede que você dê o pseudocódigo.) Embora economizemos espaço Q(mn) por esse método, o requisito de espaço auxiliar para calcular uma LCS não diminui assintoticamente, já que, de qualquer modo, precisamos do espaço Q(mn) para a tabela c.

Entretanto, podemos reduzir os requisitos de espaço assintótico para LCS-Length, já que esse procedimento só precisa de duas linhas da tabela c por vez: a linha que está sendo calculada e a linha anterior. (De fato, como o Exercício 15.4-4 pede que você mostre, podemos usar apenas um pouquinho mais que o espaço para uma linha de c para calcular o comprimento de uma LCS.) Esse aperfeiçoamento funciona se necessitamos apenas do comprimento de uma LCS; se >precisarmos reconstruir os elementos de uma LCS, a tabela menor não guardará informações suficientes para reconstituir nossas etapas no tempo $O(m + n)$.

Exercícios

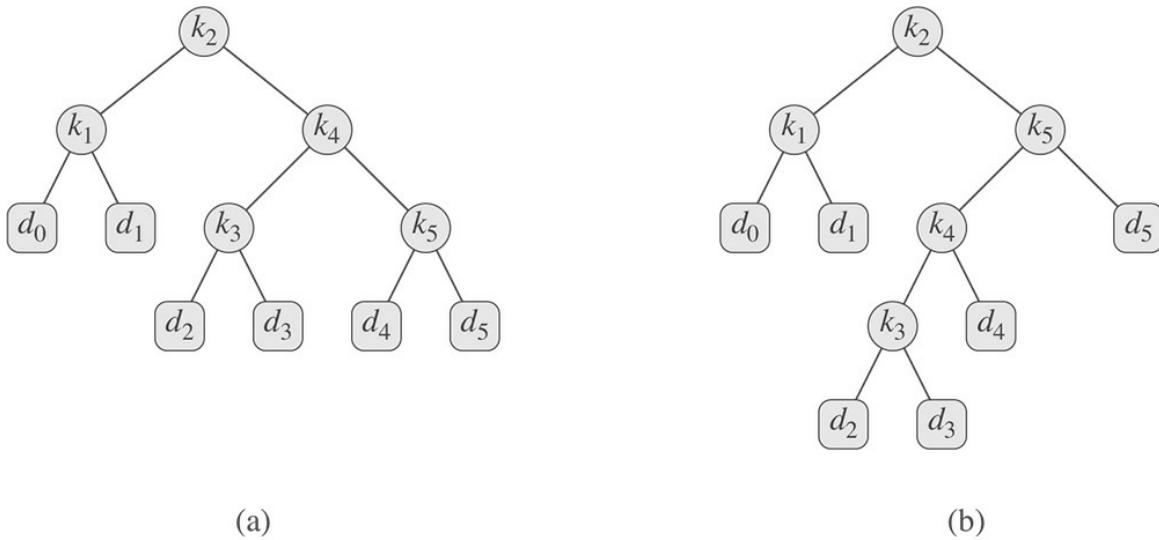
15.4-1 Determine uma LCS de $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ e $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

15.4-2 Dê o pseudocódigo para reconstruir uma LCS partindo da tabela c concluída e das sequências originais $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$ em tempo $O(m + n)$, sem usar a tabela b.

15.4-3 Dê uma versão memoizada de LCS-Length que seja executada no tempo $O(mn)$.

15.4-4 Mostre como calcular o comprimento de uma LCS usando apenas $2 \cdot \min(m, n)$ entradas na tabela c mais o espaço adicional $O(1)$. Em seguida, mostre como fazer a mesma coisa usando $\min(m, n)$ entradas mais o espaço adicional $O(1)$.

15.4-5 Dê um algoritmo de tempo $O(n_2)$ para encontrar a subsequência monotonicamente crescente mais longa de uma sequência de n números.

15.4-6 ★ Dê um algoritmo de tempo $O(n \lg n)$ para encontrar a subsequência mais longa monotonicamente crescente de uma sequência de n números. (*Sugestão*: Observe que o último elemento de uma subsequência candidata de comprimento i é no mínimo tão grande quanto o último elemento de uma subsequência candidata de comprimento $i - 1$. Mantenha as subsequências candidatas ligando-as por meio da sequência de entrada.)

15.5 ÁRVORES DE BUSCA BINÁRIA ÓTIMAS

Suponha que estejamos projetando um programa para traduzir texto do inglês para o francês. Para cada ocorrência de cada palavra inglesa no texto, precisamos procurar sua equivalente em francês. Um modo de executar essas operações de busca é construir uma árvore de busca binária com n palavras inglesas como chaves e suas equivalentes francesas como dados satélites. Como pesquisaremos a árvore para cada palavra individual no texto, queremos que o tempo total gasto na busca seja tão baixo quanto possível. Poderíamos assegurar um tempo de busca $O(\lg n)$ por ocorrência usando uma árvore vermelho-preto ou qualquer outra árvore de busca binária balanceada. Porém, as palavras aparecem com frequências diferentes, e uma palavra usada frequentemente como *the* pode aparecer longe da raiz, enquanto uma palavra raramente usada como *machicolation* apareceria perto da raiz. Tal organização reduziria a velocidade da tradução, já que o número de nós visitados durante a busca de uma chave em uma árvore de

busca binária é igual a uma unidade mais a profundidade do nó que contém a chave. Queremos que palavras que ocorrem com frequência no texto sejam colocadas mais próximas à raiz.[6] Além disso, algumas palavras no texto podem não ter nenhuma tradução para o francês[7] e, portanto, não apareceriam em nenhum lugar na árvore de busca binária. Como organizamos uma árvore de busca binária para minimizar o número de nós visitados em todas as buscas, considerando que sabemos com que frequência cada palavra ocorre?

O que precisamos é conhecido como ***árvore de busca binária ótima***. Formalmente, temos uma sequência $K = \langle k_1, k_2, ..., k_n \rangle$ de n chaves distintas em sequência ordenada (de modo que $k_1 < k_2 < ... < k_n$), e desejamos construir uma árvore de busca binária com essas chaves. Para cada chave k_i, temos uma probabilidade p_i de que uma busca seja para k_i. Algumas buscas podem ser para valores que não estão em K, e então também temos $n + 1$ "chaves fictícias" $d_0, d_1, d_2, ..., dn$ que representam valores que não estão em K. Em particular, d_0 representa todos os valores menores que k_1, d_n representa todos os valores maiores que k_n e, para $i = 1, 2, ..., n - 1$, a chave fictícia d_i representa todos os valores entre k_i e k_{i+1}. Para cada chave fictícia d_i, temos uma probabilidade q_i de que uma busca corresponda a d_i. A Figura 15.9 mostra duas árvores de busca binária para um conjunto $n = 5$ chaves. Cada chave k_i é um nó interno e cada chave fictícia d_i é uma folha. Toda busca é bem sucedida (quando encontra alguma chave k_i) ou mal sucedida (quando encontra alguma chave fictícia d_i) e, então, temos

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 . \tag{15.10}$$

i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10

Figura 15.9 Duas árvores de busca binária para um conjunto de $n = 5$ chaves com as seguintes probabilidades: (**a**) Uma árvore de busca binária com custo de busca esperado 2,80. (**b**) Uma árvore de busca binária com custo de busca esperado 2,75. Essa árvore é ótima.

Como temos probabilidades de busca para cada chave e cada chave fictícia, podemos determinar o custo esperado de uma busca em uma árvore de busca binária dada T. Vamos supor que o custo real de uma busca seja igual ao número de nós examinados, isto é, a profundidade do nó encontrado pela busca em T mais 1. Então, o custo esperado de uma busca em T é

$$\begin{aligned}\text{E[custo de busca em } T] &= \sum_{i=1}^{n}(profundidade_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n}(profundidade_T(d_i)+1)\cdot q_i \\ &= 1+\sum_{i=1}^{n} profundidade_T(k_i)\cdot p_i + \sum_{i=0}^{n} profundidade_T(d_i)\cdot q_i, \qquad (15.11)\end{aligned}$$

onde $profundidade_T$ denota a profundidade de um nó na árvore T. A última igualdade decorre da equação (15.10). Na Figura 15.9(a), podemos calcular o custo esperado da busca por nó:

nó	profundidade	probabilidade	contribuição
k_1	1	0,15	0,30
k_2	0	0,10	0,10
k_3	2	0,05	0,15
k_4	1	0,10	0,20
k_5	2	0,20	0,60
d_0	2	0,05	0,15
d_1	2	0,10	0,30
d_2	3	0,05	0,20
d_3	3	0,05	0,20
d_4	3	0,05	0,20
d_5	3	0,10	0,40
Total			2,80

Para um dado conjunto de probabilidades, queremos construir uma árvore de busca binária cujo custo de busca esperado seja o menor de todos. Damos a essa árvore o nome de ***árvore de busca binária ótima***. A Figura 15.9(b) mostra uma árvore de busca binária ótima para as probabilidades dadas na legenda da figura; seu custo esperado é 2,75. Esse exemplo mostra que uma árvore de busca binária ótima não é necessariamente uma árvore cuja altura global seja a menor. Nem funciona necessariamente construir uma árvore de busca binária ótima sempre colocando a chave com maior probabilidade na raiz. Aqui, a chave k_5 tem a maior probabilidade de busca de qualquer chave, entretanto a raiz da árvore de busca binária ótima mostrada é k_2. (O custo esperado mais baixo de qualquer árvore de busca binária com k_5 na raiz é 2,85.)

Como ocorre com a multiplicação de cadeias de matrizes, a verificação exaustiva de todas as possibilidades não produz um algoritmo eficiente. Podemos identificar os nós de qualquer árvore binária de n nós com as chaves k_1, k_2,..., k_n para construir uma árvore de busca binária e depois adicionar as chaves fictícias como folhas. No Problema 12-4, vimos que o número de árvores binárias com n nós é $(4^n/n_{3/2})$ e, portanto, em uma busca exaustiva teríamos de examinar um número exponencial de árvores de busca binária. Então, não é nenhuma surpresa que resolvamos esse problema com programação dinâmica.

Etapa 1: A estrutura de uma árvore de busca binária ótima

Para caracterizar a subestrutura ótima de árvores de busca binária ótima, começamos com uma observação sobre subárvores. Considere qualquer subárvore de uma árvore de busca binária. Ela deve conter chaves em uma faixa contígua k_i,...,k_j, para algum $1 \leq i \leq j \leq n$. Além disso, uma subárvore que contém chaves k_i,...,k_j também deve ter como folhas as chaves fictícias d_i-1,...,d_j.

Agora podemos expressar a subestrutura ótima: se uma árvore de busca binária ótima T tem uma subárvore T' que contém chaves k_i,...,k_j, então essa subárvore T' deve também ser ótima para o subproblema com chaves k_i,...,k_j e

chaves fictícias $d_{i-1},...,d_j$. O argumento habitual de recortar e colar é aplicável. Se houvesse uma subárvore T'' cujo custo esperado fosse mais baixo que o de T', poderíamos recortar T' de T e colar T'' no seu lugar, resultando em uma árvore de busca binária de custo esperado mais baixo que T, contradizendo assim a otimalidade de T.

Precisamos usar a subestrutura ótima para mostrar que podemos construir uma solução ótima para o problema partindo de soluções ótimas para subproblemas. Dadas as chaves $k_i,...,k_j$, uma dessas chaves, digamos k_r ($i \le r \le j$), é a raiz de uma subárvore ótima que contém essas chaves. A subárvore esquerda da raiz k_r contém as chaves $k_i,...,k_{r-1}$ (e chaves fictícias $d_{i-1},...,d_{r-1}$), e a subárvore direita contém as chaves $k_{r+1},...,k_j$ (e chaves fictícias $d_r,...,d_j$). Desde que examinemos todas as raízes candidatas k_r, onde $i \le r \le j$, e determinemos todas as árvores de busca binária ótimas que contêm $k_i,...,k_{r-1}$ e as que contêm $k_{r+1},...,k_j$, é garantido que encontraremos uma árvore de busca binária ótima.

Há um detalhe que vale a pena observar sobre subárvores "vazias". Suponha que em uma subárvore com chaves $k_i,...,k_j$, selecionemos k_i como a raiz. Pelo argumento que acabamos de apresentar, a subárvore esquerda de k_i contém as chaves $k_i,...,k_{i-1}$. Interpretamos que essa sequência não contém nenhuma chave. Contudo, lembre-se de que subárvores também contêm chaves fictícias. Adotamos a seguinte convenção: uma subárvore que contém chaves $k_i,...,k_{i-1}$ não tem nenhuma chave real, mas contém a única chave fictícia d_{i-1}. Simetricamente, se selecionarmos k_j como a raiz, a subárvore direita de k_j contém as chaves $k_{j+1},...,k_j$; essa subárvore direita não contém nenhuma chave real, mas contém a chave fictícia d_j.

Etapa 2: Uma solução recursiva

Estamos prontos para definir o valor de uma solução ótima recursivamente. Escolhemos, como domínio de nosso subproblema, encontrar uma árvore de busca binária ótima que contenha as chaves $k_i,...,k_j$, onde $i \ge 1$, $j \le n$ e $j \ge i - 1$. (Quando $j = i - 1$ não existe nenhuma chave real; temos apenas a chave fictícia d_{i-1}.) Vamos definir $e[i, j]$ como o custo esperado de pesquisar uma árvore de busca binária ótima que contenha as chaves $k_i,...,k_J$. Em última análise, desejamos calcular $e[1, n]$.

O caso fácil ocorre quando $j = i - 1$. Então, temos apenas a chave fictícia d_{i-1}. O custo de busca esperado é $e[i, i - 1] = q_{i-1}$.

Quando $j \ge i$, precisamos selecionar uma raiz k que esteja entre $k_i,...,kj$ e fazer de uma árvore de busca binária ótima com chaves $k_i,...,k_{r-1}$ sua subárvore esquerda e de uma árvore de busca binária ótima com chaves $k_{r+1},...,k_j$ sua subárvore direita. O que acontece com o custo de busca esperado de uma subárvore quando ela se torna uma subárvore de um nó? A profundidade de cada nó na subárvore aumenta de 1. Pela equação (15.11), o custo de busca esperado dessa subárvore aumenta de uma quantidade igual à soma de todas as probabilidades na subárvore. Para uma subárvore com chaves $k_i,...,k_j$, vamos denotar essa soma de probabilidades como

$$w(i,j) = \sum_{l=i}^{j} p_1 + \sum_{l=i-1}^{j} q_1. \qquad (15.12)$$

Assim, se k_r é a raiz de uma subárvore ótima contendo chaves $k_i,...,k_j$, temos

$$e[i,j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)).$$

Observando que

$$w(i,j) = w(i, r-1) + p_r + w(r+1, j),$$

reescrevemos $e[i, j]$ como

$$e[i,j] = w[i, r-1] + e[r+1, j] + w(i,j). \qquad (15.13)$$

A equação recursiva (15.13) pressupõe que sabemos qual nó k_r usar como raiz. Escolhemos a raiz que dá o custo de busca esperado mais baixo, o que dá nossa formulação recursiva final:

$$e[i,j] = \begin{cases} q_{i-1} & \text{se } j = i-1, \\ \min_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{se } i \le j. \end{cases} \quad (15.14)$$

Os valores $e[i, j]$ dão os custos de busca esperados em árvores de busca binária ótimas. Para ajudar a controlar a estrutura de árvores de busca binária ótimas, definimos $raiz[i, j]$, para $1 \le i \le j \le n$, como o índice r para o qual k_r é a raiz de uma árvore de busca binária ótima contendo chaves $k_i,...,k_j$. Veremos como calcular os valores de $raiz[i, j]$, mas deixamos a construção da árvore de busca binária ótima com esses valores para o Exercício 15.5-1.

Etapa 3: Cálculo do custo de busca esperado de uma árvore de busca binária ótima

Até aqui, você deve ter notado algumas semelhanças entre as caracterizações que fizemos de árvores de busca binária ótimas e multiplicação de cadeias de matrizes. Para ambos os domínios de problemas, nossos subproblemas consistem em subfaixas de índices contíguos. Uma implementação recursiva direta da equação (15.14) seria tão ineficiente quanto um algoritmo recursivo direto de multiplicação de cadeias de matrizes. Em vez disso, armazenamos os valores $e[i, j]$ em uma tabela $e[1 .. n + 1, 0 .. n]$. O primeiro índice precisa ir até $n + 1$ em vez de n porque, para ter uma subárvore contendo apenas a chave fictícia d_n, precisaremos calcular e armazenar $e[n + 1, n]$. O segundo índice tem de começar de 0 porque, para ter uma subárvore contendo apenas a chave fictícia d_0, precisaremos calcular e armazenar $e[1, 0]$. Usamos somente as entradas $e[i, j]$ para as quais $j \ge i - 1$. Empregamos também uma tabela $raiz[i, j]$ para registrar a raiz da subárvore que contém as chaves $k_i,...,k_j$. Essa tabela utiliza somente as entradas para as quais $1 \le i \le j \le n$.

Precisaremos de uma outra tabela para eficiência. Em vez de calcular o valor de $w(i, j)$ desde o início toda vez que estamos calculando $e[i, j]$ — o que exigiria $Q(j - i)$ adições —, armazenamos esses valores em uma tabela $w[1..n + 1, 0..n]$. Para o caso-base, calculamos $w[i, i - 1] = q_i{-1}$ para $1 \le i \le n$. Para $j \ge i$, calculamos

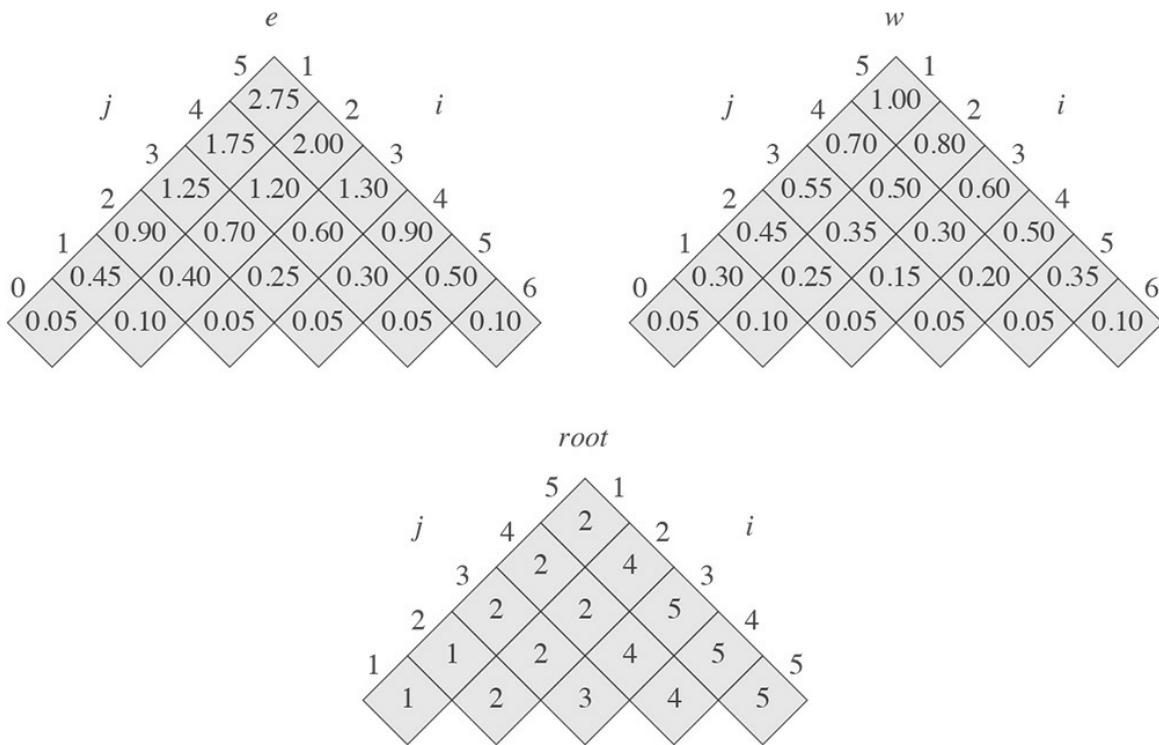
$$w[i,j] = w[i,j-1] + p_j + q_j\,. \quad (15.15)$$

Assim, podemos calcular cada um dos $Q(n_2)$ valores de $w[i, j]$ no tempo $Q(1)$.

O pseudocódigo a seguir, toma como entradas as probabilidades $p_1,...,p_n$ e $q_0,...,q_n$ e o tamanho n, e retorna as tabelas e e $raiz$.

OPTIMAL-BST (p, q, n)

```
1   sejam e[1..n + 1,0..n], w[1..n + 1, 0..n],
            e raiz [1..n, 1..n] tabelas novas
2   for i = 1 to n + 1
3       e[i, i − 1] = q_{i-1}
4       w[i, i − 1] = q_{i-1}
5   for l = 1 to n
6       for i = 1 to n − l + 1
7           j = i + l − 1
8           e[i, j] = ∞
9           w[i, j] = w[i, j − 1] + p_j + q_j
10          for r = i to j
11              t = e[i, r − 1] + e[r + 1, j] + w[i, j]
12              if t < e[i, j]
13                  e[i, j] = t
14                  raiz[i, j] = r
15   return e, raiz
```

Pela descrição anterior e pela semelhança com o procedimento MATRIX-CHAIN-ORDER da Seção 15.2, você constatará que a operação desse procedimento é razoavelmente direta. O laço **for** das linhas 2-4 inicializa os valores de $e[i, i - 1]$ e $w[i, i - 1]$. Então, o laço **for** das linhas 5-14 usa as recorrências (15.14) e (15.15) para calcular $e[i, j]$ e $w[i, j]$ para $1 \leq i \leq j \leq n$. Na primeira iteração, quando $l = 1$, o laço calcula $e[i, i]$ e $w[i, i]$ para $i = 1, 2,...,n$. A segunda iteração, com $l = 2$, calcula $e[i, i + 1]$ e $w[i, i + 1]$ para $i = 1, 2,...,n - 1$, e assim por diante. O laço **for** mais interno, nas linhas 10-14, experimenta cada índice candidato r para determinar que chave k_r usar como raiz de uma árvore de busca binária ótima contendo chaves $k_i,...,k_j$. Esse laço **for** salva o valor atual do índice r em $raiz[i, j]$ sempre que encontra uma chave melhor para usar como raiz.

A Figura 15.10 mostra as tabelas $e[i, j]$, $w[i, j]$ e $raiz[i, j]$ calculadas pelo procedimento OPTIMAL-BST para a distribuição de chaves mostrada na Figura 15.9. Como no exemplo de multiplicação de cadeias de matrizes da Figura 15.5, as tabelas sofreram uma rotação para colocar a diagonal principal na posição horizontal. OPTIMAL-BST calcula as linhas de baixo para cima e da esquerda para a direita dentro de cada linha.

O procedimento OPTIMAL-BST demora o tempo $Q(n_3)$, exatamente como MATRIX-CHAIN-ORDER. É fácil verificar que o tempo de execução é $O(n_3)$, já que seus laços **for** estão aninhados em profundidade três e cada índice de laço exige no máximo n valores. Os índices de laços em OPTIMAL-BST não têm exatamente os mesmos limites que os de MATRIX-CHAIN-ORDER, mas eles estão abrem no máximo 1 em todas as direções. Assim, exatamente como MATRIX-CHAIN-ORDER, o procedimento OPTIMAL-BST demora o tempo (n_3).

Exercícios

15.5-1 Escreva o pseudocódigo para o procedimento CONSTRCUT-OPTIMAL-BST(*raiz*) que, dada a tabela *raiz*, dê como saída a estrutura de uma árvore de busca binária ótima. No exemplo da Figura 15.10, seu procedimento deve imprimir a estrutura

Figura 15.10 As tabelas ei, j , wi, j e $raizi, j$ calculadas por OPTIMAL-BST para a distribuição de chaves mostrada na Figura 15.9. As tabelas sofreram uma rotação para colocar as diagonais principais na posição horizontal.

k_2 é a raiz

k_1 é o filho à esquerda de k_2

d_0 é o filho à esquerda de k_1

d_1 é o filho à direita de k_1

k_5 é o filho à direita de k_2

k_4 é o filho à esquerda de k_5

k_3 é o filho à esquerda de k_4

d_2 é o filho à esquerda de k_3

d_3 é o filho à direita de k_3

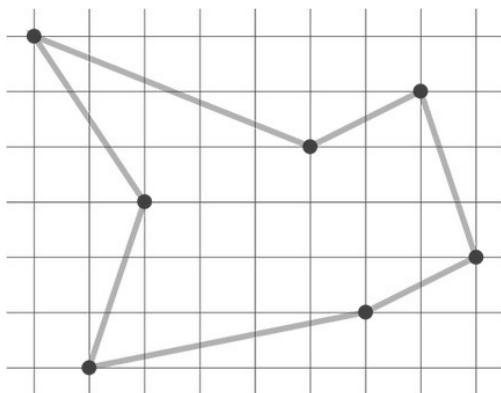
d_4 é o filho à direita de k_4

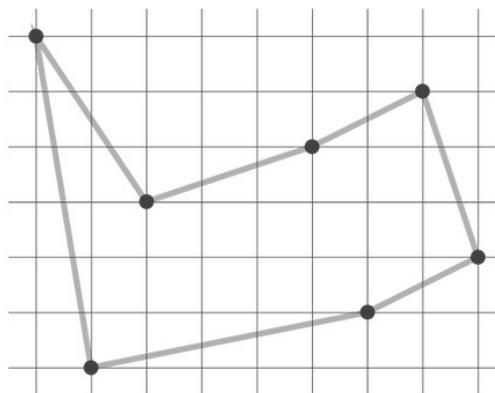
d_5 é o filho à direita de k_5

correspondente à árvore de busca binária ótima mostrada na Figura 15.9(b).

15.5-2 Determine o custo e a estrutura de uma árvore de busca binária ótima para um conjunto de $n = 7$ chaves com as seguintes probabilidades:

i	0	1	2	3	4	5	6	7
p_i		0,04	0,06	0,08	0,02	0,10	0,12	0,14
q_i	0,06	0,06	0,06	0,06	0,05	0,05	0,05	0,05

15.5-3 Suponha que, em vez de manter a tabela $w[i, j]$, calculássemos o valor de $w(i, j)$ diretamente da equação (15.12) na linha 9 de Optimal-BST e utilizássemos esse valor calculado na linha 11. Como essa mudança afetaria o tempo de execução assintótico de Optimal-BST?

15.5-4 ★Knuth [214] mostrou que sempre existem raízes de subárvores ótimas tais que $raiz[i, j-1] \leq raiz[i, j] \leq raiz[i+1, j]$ para todo $1 \leq i < j \leq n$. Use esse fato para modificar o procedimento Optimal-BST de modo que ele seja executado no tempo $Q(n_2)$.

Problemas

15.1 Caminho simples mais longo em um grafo acíclico dirigido

Suponha que tenhamos um grafo acíclico dirigido $G = (V, E)$ com pesos de arestas com valores reais e dois vértices distinguidos s e t. Descreva uma abordagem de programação dinâmica para determinar um caminho simples ponderado mais longo de s a t. Qual é a aparência do grafo de subproblema? Qual é a eficiência do seu algoritmo?

Figura 15.11 Sete pontos no plano, mostrados sobre uma grade unitária. **(a)** O caminho fechado mais curto, com comprimento aproximado de 24,89. Esse caminho não é bitônico. **(b)** O caminho fechado bitônico mais curto para o mesmo conjunto de pontos. Seu comprimento é aproximadamente 25,58.

15.2 Subsequência palíndromo mais longa

Um ***palíndromo*** é uma cadeia não vazia em algum alfabeto que é lida do mesmo modo da esquerda para a direita ou da direita para a esquerda. Exemplos de palíndromos são todas cadeias de comprimento 1, radar, asa, reter e oco.

Dê um algoritmo eficiente para encontrar o palíndromo mais longo que é uma subsequência de uma cadeia de entrada dada. Por exemplo, dada a entrada character nosso algoritmo retornaria carac. Qual é o tempo de execução do seu algoritmo?

15.3 ***Problema do caixeiro-viajante euclidiano bitônico***:

No ***problema do caixeiro-viajante euclidiano*** temos um conjunto de n pontos no plano e queremos determinar o caminho fechado mais curto que conecta todos os n pontos. A Figura 15.11(a) mostra a solução para um problema de sete pontos. O problema geral é NP-difícil e, portanto, acredita-se que sua solução requeira mais que tempo polinomial (ver Capítulo 34).

J. L. Bentley sugeriu que simplificássemos o problema restringindo nossa atenção a ***caminhos fechados bitônicos***, isto é, caminhos fechados que começam no ponto da extrema esquerda, seguem estritamente da esquerda para a direita até o ponto da extrema direita e depois voltam estritamente da direita para a esquerda até o ponto de partida. A Figura 15.11(b) mostra o caminho fechado bitônico mais curto para os mesmos sete pontos. Nesse caso, é possível um algoritmo de tempo polinomial.

Descreva um algoritmo de tempo $O(n_2)$ para determinar um caminho fechado bitônico ótimo. Você pode considerar que não existem dois pontos com a mesma coordenada x. (*Sugestão*: Desloque-se da esquerda para a direita, mantendo possibilidades ótimas para as duas partes do caminho fechado.)

15.4 ***Como obter uma impressão nítida***

Considere o problema de obter, em uma impressora, uma impressão nítida de um parágrafo em fonte monoespaçada (todos os caracteres têm a mesma largura). O texto de entrada é uma sequência de n palavras de comprimentos l_1, l_2, ... l_n, medidos em caracteres. Queremos imprimir esse parágrafo nitidamente em uma série de linhas que contêm no máximo M caracteres cada uma. Nosso critério de "nitidez" é dado a seguir. Se determinada linha contém palavras de i até j, onde $i \leq j$, e deixamos exatamente um espaço entre as palavras, o número de caracteres de espaço extras no final da linha é $M - j + i - \sum_{k=1}^{j} l_k$, que deve ser não negativo para que as palavras caibam na linha. Desejamos minimizar a soma em todas as linhas, exceto a última, dos cubos dos números de caracteres de espaço extras nas extremidades das linhas. Dê um algoritmo de programação dinâmica para imprimir um parágrafo de n palavras nitidamente em uma impressora. Analise o tempo de execução e os requisitos de espaço do seu algoritmo.

15.5 ***Distância de edição***

Para transformar uma cadeia de texto de origem $x[1..m]$ na cadeia de texto que desejamos $y[1..n]$, podemos executar várias operações de transformação. Nossa meta é, dados x e y, produzir uma série de transformações que mudam x para y. Usamos um arranjo z — considerado grande o suficiente para conter todos os caracteres de que precisará — para conter os resultados intermediários. Inicialmente, z está vazio e, no término, devemos ter $z[j] = y[j]$ para $j = 1, 2, ..., n$. Mantemos índices atuais i em x e j em z, e as operações podem alterar z e esses índices. Inicialmente, $i = j = 1$. Temos de examinar cada caractere em x durante a transformação, o que significa que no fim da sequência de operações de transformação devemos ter $i = m + 1$.

Podemos escolher entre seis operações de transformação:

Copiar um caractere de x para z fazendo $z[j] = x[i]$ e incrementar i e j. Essa operação examina $x[i]$.

Substituir um caractere de x por outro caractere c fazendo $z[j] = c$ e incrementar i e j. Essa operação examina $x[i]$.

Excluir um caractere de x incrementando i, mas deixando j inalterado. Essa operação examina $x[i]$.

Inserir o caractere c em z definindo $z[j] = c$ e incrementar j, mas deixar i inalterado. Essa operação não examina nenhum caractere de x.

Transpor os dois caracteres seguintes, copiando-os de x para z mas na ordem oposta; para tal fazemos $z[j] = x[i + 1]$ e $z[j + 1] = x[i]$, e fazemos $i = i + 2$ e $j = j + 2$. Essa operação examina $x[i]$ e $x[i + 1]$.

Eliminar o restante de x fazendo $i = m + 1$. Essa operação examina todos os caracteres em x que ainda não foram examinados. Essa operação, se executada, deverá ser a última.

Como exemplo, um modo de transformar a corrente de origem algorithm na corrente desejada altruistic é usar a sequência de operações a seguir, onde os caracteres sublinhados são $x[i]$ e $z[j]$ após a operação:

Operação	x	z
cadeias iniciais	algorithm	_
copiar	algorithm	a_
copiar	algorithm	al_
substituir por *t*	algorithm	alt_
excluir	algorithm	alt_
copiar	algorithm	altr_
inserir *u*	algorithm	altru_
inserir *i*	algorithm	altrui_
inserir *s*	algorithm	altruis_
transpor	algorithm	altruisti_
inserir *c*	algorithm	altruistic_
eliminar	algorithm_	altruistic_

Observe que há várias outras sequências de operações de transformação que convertem algorithm em altruistic.

Cada uma das operações de transformação tem um custo associado. O custo de uma operação depende da aplicação específica, mas consideramos que o custo de cada operação é uma constante que conhecemos. Supomos também que os custos individuais das operações copiar e substituir são menores que os custos combinados das operações excluir e inserir, senão as operações de copiar e substituir não seriam usadas. O custo de uma dada sequência de operações de transformação é a soma dos custos das operações individuais na sequência. Para a sequência que estudamos neste exercício, o custo de transformar algorithm em altruistic é

(3 · custo(copiar)) + custo(substituir) + custo(excluir) + (4 · custo(inserir)) + custo(girar) + custo(eliminar).

a. Dadas duas sequências $x1..m$ e $y1..n$ e um conjunto de custos de operação, a ***distância de edição*** de x para y é o custo da sequência menos dispendiosa de operações que transforma x em y. Descreva um algoritmo de programação dinâmica que determine a distância de edição de $x1..m$ para $y1..n$ e imprime uma sequência de operações ótima. Analise o tempo de execução e os requisitos de espaço de seu algoritmo.

O problema da distância de edição generaliza o problema de alinhar duas sequências de DNA (veja, por exemplo, Setubal e Meidanis [310, Seção 3.2]). Há vários métodos para medir a semelhança entre duas sequências de DNA por alinhamento. Um dos métodos para alinhar duas sequências x e y consiste em inserir

espaços em posições arbitrárias nas duas sequências (inclusive em qualquer extremidade) de modo que as sequências resultantes x' e y' tenham o mesmo comprimento, mas não um espaço na mesma posição (isto é, para nenhuma posição j, $x'[j]$ e $y'[j]$ são espaços). Então, atribuímos uma "pontuação" a cada posição. A posição j recebe uma pontuação da seguinte maneira:

- + 1 se $x'[j] = y'[j]$ e nenhum deles é um espaço,
- -1 se $x'j \neq y'j$ e nenhum deles é um espaço,
- -2 se $x'j$ ou $y'j$ é um espaço.

A pontuação para o alinhamento é a soma das pontuações das posições individuais. Por exemplo, dadas as sequências x = GATCGGCAT e y = CAATGTGAATC, um alinhamento é

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

Um + sob uma posição indica uma pontuação +1 para aquela posição, um – indica a pontuação -1 e um * indica a pontuação -2; portanto, esse alinhamento tem uma pontuação total igual a $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

b. Explique como expressar o problema de determinar um alinhamento ótimo como um problema de distância de edição usando um subconjunto das operações de transformação copiar, substituir, excluir, inserir, girar e eliminar.

15.6 Planejamento de uma festa da empresa

O professor Stewart presta consultoria ao presidente de uma corporação que está planejando uma festa da empresa. A empresa tem uma estrutura hierárquica, isto é, as relações entre os supervisores formam uma árvore com raiz no presidente. O pessoal do escritório classificou cada funcionário segundo uma avaliação de sociabilidade que é um número real. Para tornar a festa divertida para todos os participantes, o presidente não quer que um funcionário e seu supervisor imediato participem.

O professor Stewart recebe a árvore que descreve a estrutura da corporação usando a representação de filho à esquerda, irmão à direita descrita na Seção 10.4. Cada nó da árvore contém, além dos ponteiros, o nome de um funcionário e o posto que ele ocupa na escala de classificação de sociabilidade. Descreva um algoritmo para compor uma lista de convidados que maximize a soma das avaliações de sociabilidade dos convidados. Analise o tempo de execução do seu algoritmo.

15.7 Algoritmo de Viterbi

Podemos usar programação dinâmica em um grafo dirigido $G = (V, E)$ para reconhecimento de voz. Cada aresta $(u, v) \in E$ é identificada por um som (u, v) de um conjunto finito Σ de sons. O grafo rotulado é um modelo formal de uma pessoa falando uma linguagem restrita. Cada caminho no grafo que parte de um vértice distinto $v_0 \in V$ corresponde a uma sequência possível de sons produzidos pelo modelo. Definimos o rótulo de um caminho dirigido como a concatenação dos rótulos das arestas nesse caminho.

a. Descreva um algoritmo eficiente que, dado um grafo com arestas rotuladas G contendo um vértice distinto v_0 e uma sequência $s = \langle s_1, s_2, ..., s_k \rangle$ de sons pertencentes ao conjunto Σ, retorne um caminho em

G que começa em v_0 e tenha s como rótulo, se tal caminho existir. Caso contrário, o algoritmo deve retornar NO-SUCH-PATH. Analise o tempo de execução de seu algoritmo. (*Sugestão*: Os conceitos do Capítulo 22 poderão ser úteis.)

Agora, suponha que toda aresta $(u, v) \in E$ tenha uma probabilidade associada não negativa $p(u, v)$ de percorrer a aresta (u, v) desde o vértice u e, assim, produzir o som correspondente. A soma das probabilidades das arestas que saem de qualquer vértice é igual a 1. A probabilidade de um caminho é definida como o produto das probabilidades de suas arestas. Podemos considerar a probabilidade de um caminho que começa em v_0 como a probabilidade de um "percurso aleatório" começando em v_0 seguir o caminho especificado, onde escolhemos aleatoriamente qual aresta que sai de um vértice u tomar de acordo com as probabilidades das arestas disponíveis que partem de u.

b. Amplie sua resposta à parte (a), de modo que, se um caminho for retornado, ele é um *caminho mais provável* que começa em v_0 e tem rótulo s. Analise o tempo de execução do seu algoritmo.

15.8 *Compressão de imagem por descostura* (seam carving)

Temos uma figura em cores que consiste em um arranjo mn $A[1..m, 1..n]$ de pixels, onde cada pixel especifica uma tripla de intensidades de vermelho, verde e azul (RGB). Suponha que queiramos comprimir ligeiramente essa figura. Especificamente, queremos remover um pixel de cada uma das m linhas, de modo que a figura inteira fique um pixel mais estreita. Porém, para evitar distorção nos efeitos visuais, é necessário que os pixels removidos em duas linhas adjacentes estejam na mesma coluna ou em colunas adjacentes. Os pixels removidos formam uma "costura" da linha superior até a linha inferior. Nessa costura, os pixels sucessivos são adjacentes na vertical e na diagonal.

a. Mostre que o número de tais costuras possíveis cresce no mínimo exponencialmente em m, considerando que $n > 1$.

b. Agora suponha que juntamente com cada pixel Ai, j, calculamos uma medida de distorção de valor real di, j, que indica qual seria o grau de distorção causado pela remoção do pixel Ai, j. Intuitivamente, quanto mais baixa a medida de distorção causada por um pixel, mais semelhante a seus vizinhos é esse pixel. Suponha ainda que definimos a medida de distorção de uma costura como a soma das medidas de distorção de seus pixels.

Dê um algoritmo para encontrar uma costura que tenha a medida de distorção mais baixa. Qual seria a eficiência desse algoritmo?

15.9 *Quebra de cadeia*

Certa linguagem de processamento de cadeias permite que um programador quebre uma cadeia em dois pedaços. Como essa operação copia a cadeia, quebrar uma cadeia de n caracteres em dois pedaços tem um custo de n unidades de tempo. Suponha que um programador queira quebrar uma cadeia em muitos pedaços. A ordem em que as quebras ocorrem pode afetar a quantidade total de tempo gasto. Por exemplo, suponha que a programadora queira quebrar uma cadeia de 20 caracteres depois dos caracteres 2, 8 e 10 (numerando os caracteres em ordem ascendente a partir da extremidade esquerda e começando de 1). Se ela programar as quebras da esquerda para a direita, a primeira quebra custará 20 unidades de tempo, a segunda quebra custará 18 unidades de tempo (quebra da cadeia dos caracteres 3 a 20 no caractere 8), e a terceira quebra custará 12 unidades de tempo, totalizando 50 unidades de tempo. Entretanto, se ela programar as quebras da direita para a esquerda, a primeira quebra custará 20 unidades de tempo, a segunda quebra custará 10 unidades de tempo, e a terceira quebra custará 8 unidades de tempo, totalizando 38 unidades de tempo.

Ainda em uma outra ordem, ela poderia programar a primeira quebra em 8 (custo 20), quebrar o pedaço à esquerda em 2 (custo 8) e, finalmente, o pedaço à direita em 10 (custo 12), o que dá um custo total de 40. Projete um algoritmo que, dados os números de posição dos caracteres após os quais ocorrerão as quebras, determine uma sequência de menor custo dessas quebras. Mais formalmente, dada uma cadeia S com n caracteres e um arranjo $L[1..m]$ que contém os pontos de quebra, calcule o menor custo para uma sequência de quebras juntamente com uma sequência de quebras que atinja esse custo.

15.10 Planejamento de uma estratégia de investimento

Como você conhece bem algoritmos, consegue um emprego interessante na Acme Computer Company, além de um bônus contratual de $10.000. Você decide investir esse dinheiro com o objetivo de maximizar o retorno em 10 anos. Então, decide contratar a Amalgamated Investment Company para gerenciar os seus investimentos. Essa empresa exige que você observe as regras descritas a seguir. Ela oferece n investimentos diferentes, numerados de 1 a n. Em cada ano j, o investimento dá uma taxa de retorno de r_{ij}. Em outras palavras, se você investiu d dólares no investimento i no ano j, ao final do ano j terá dr_{ij} dólares. As taxas de retorno são garantidas, isto é, a empresa informa todas as taxas de retorno para os próximos 10 anos para cada investimento. Você decide o rumo de seus investimentos uma vez por ano. Ao fim de cada ano, você pode deixar o dinheiro ganho no ano anterior nos mesmos investimentos, ou pode transferir dinheiro para outros investimentos, seja por transferência entre investimentos existentes, ou seja por transferência para um novo investimento. Se você não movimenta o dinheiro entre dois anos consecutivos, você paga uma taxa de f_1 dólares, enquanto que se houver transferência, você paga uma taxa de f_2 dólares, onde $f_2 > f_1$.

a. O problema, como enunciado, permite que você aplique seu dinheiro em vários investimentos a cada ano. Prove que existe uma estratégia de investimento ótima que, a cada ano, investe todo o dinheiro em um único investimento. (Lembre-se de que uma estratégia de investimento ótima maximiza a quantia investida após 10 anos e não se preocupa com outros objetivos, como minimizar riscos.)

b. Prove que o problema de planejar sua estratégia de investimentos ótima exibe subestrutura ótima.

c. Projete um algoritmo que planeje sua estratégia de investimentos ótima. Qual é o tempo de execução desse algoritmo?

d. Suponha que a Amalgamated Investments tenha imposto a seguinte restrição adicional: a qualquer instante, você não pode ter mais de $15.000 em qualquer dos investimentos. Mostre que o problema de maximizar sua receita ao final de 10 anos deixa de exibir subestrutura ótima.

15.11 Planejamento de estoque

A Ricky Dink Company fabrica máquinas para restaurar a superfície de rinques de patinação no gelo. A demanda por tais produtos varia de mês a mês e, por isso, a empresa precisa desenvolver uma estratégia de planejamento de produção dada a demanda flutuante, porém previsível. A empresa quer projetar um plano para os próximos n meses e sabe qual é a demanda d_i para cada mês i, isto é, o número de máquinas que venderá nesse mês. Seja $D = \sum_{i=1}^{n} d_i$ a demanda total para os próximos n meses. A empresa mantém um quadro permanente de funcionários de tempo integral que fornecem a mão de obra para produzir até m máquinas por mês. Se ela precisar fabricar mais de m máquinas em determinado mês, pode contratar mão de obra temporária adicional, a um custo calculado de c dólares por máquina. Além disso, se ao final de um mês a empresa tiver em estoque qualquer número de máquinas não vendidas terá de pagar custos de estoque. O custo de estocar j máquinas é dado como uma função $h(j)$ para $j = 1, 2,..., D$, onde $h(j) \geq 0$

para $1 \leq j \leq D$ e $h(j) \leq h(j + 1)$ para $1 \leq j \leq D - 1$.

Dê um algoritmo para calcular um plano de produção para a empresa que minimize seus custos e, ao mesmo tempo, atenda à demanda. O tempo de execução deve ser polinomial em n e D.

15.12 Contratação de jogadores de beisebol donos de seu passe

Suponha que você seja o gerente geral de um time de beisebol da primeira divisão. No período entre temporadas, você precisa contratar para sua equipe alguns jogadores donos de seu próprio passe. O dono do time disponibilizou $X para gastar com esses jogadores e você pode gastar menos de $X no total, mas será demitido se gastar mais de $X. Você está considerando N posições diferentes e, para cada posição, há P jogadores disponíveis.[8] Como não quer sobrecarregar seu plantel com muitos jogadores em alguma posição, você decide contratar no máximo um jogador reserva adicional para cada posição. (Se não contratar nenhum jogador para uma determinada posição, você planeja continuar apenas com os jogadores de seu time titular para tal posição.)

Para determinar o valor futuro de um jogador, você decide usar uma estatística sabermétrica[9] conhecida como VORP (*value over replacement player* — valor de um reserva). Um jogador que tenha um VORP mais alto é mais valioso que um jogador com VORP mais baixo. Contratar um jogador que tenha um VORP mais alto não é necessariamente mais caro que contratar um com VORP mais baixo porque há outros fatores que determinam o custo do contrato, além do valor do jogador.

Para cada jogador reserva, você tem três informações:

- a posição do jogador,
- quanto custará contratar o jogador e
- o VORP do jogador.

Projete um algoritmo que maximize o VORP total dos jogadores que você contrata e, ao mesmo tempo, não gaste mais de $X no total. Suponha que o contrato de cada jogador seja sempre um múltiplo de $100.000. Seu algoritmo deve dar como saída o VORP total dos jogadores contratados, o total de dinheiro gasto e uma lista dos jogadores contratados. Analise o tempo de execução e o requisito de espaço do seu algoritmo.

NOTAS DO CAPÍTULO

R. Bellman começou o estudo sistemático de programação dinâmica em 1955. A palavra "programação", tanto aqui quanto em programação linear, se refere ao uso de um método de solução tabular. Embora as técnicas de otimização que incorporam elementos de programação dinâmica fossem conhecidas antes, Bellman deu à área uma sólida base matemática [37].

Galil e Park [125] classificam algoritmos de programação dinâmica de acordo com o tamanho da tabela e o número de outras entradas de tabela das quais cada entrada depende. Eles denominam um algoritmo de programação dinâmica tD/eD se o tamanho de sua tabela for $O(n^t)$ e cada entrada depender de outras $O(n^e)$ entradas. Por exemplo, o algoritmo de multiplicação de cadeia de matrizes na Seção 15.2 seria $2D/1D$, e o algoritmo da subsequência comum mais longa na Seção 15.4 seria $2D/0D$.

Hu e Shing [182, 183] apresentam um algoritmo de tempo $O(n \lg n)$ para o problema de multiplicação de cadeias de matrizes.

O algoritmo de tempo $O(mn)$ para o problema da subsequência comum mais longa parece ser um algoritmo folclórico. Knuth [70] levantou a questão da existência ou não de algoritmos subquadráticos para o problema da LCS. Masek e Paterson [244] responderam afirmativamente a essa pergunta, dando um algoritmo que é executado no tempo $O(mn/\lg n)$, onde $n \leq m$ e as sequências são extraídas de um conjunto de tamanho limitado. Para o caso especial no qual nenhum elemento aparece mais de uma vez em uma sequência de entrada, Szymanski [326] mostra como resolver o problema no tempo $O((n + m)\lg(n + m))$. Muitos desses resultados se estendem ao problema de calcular distâncias de edição de cadeias (Problema 15-5).

Um artigo anterior sobre codificações binárias de comprimento variável apresentado por Gilbert e Moore [133] teve aplicações na construção de árvores de busca binária ótimas para o caso no qual todas as probabilidades p_i sejam 0; esse artigo contém um algoritmo de tempo $O(n_3)$. Aho, Hopcroft e Ullman [5] apresentam o algoritmo da Seção 15.5. O Exercício 15.5-4 se deve a Knuth [212]. Hu e Tucker [184] criaram um algoritmo para o caso no qual todas as probabilidades p_i sejam 0 e utiliza o tempo $O(n_2)$ e o espaço $O(n)$; mais tarde, Knuth [211] reduziu o tempo para $O(n \lg n)$.

O Problema 15-8 se deve a Avidan e Shamir [27], que apresentaram na Web um maravilhoso vídeo que ilustra essa técnica de compressão de imagem.

---

1 Se exigíssemos que as peças fossem cortadas em ordem não decrescente de tamanho, haveria um número menor de modos a considerar. Para $n = 4$, consideraríamos somente cinco desses modos: partes (a), (b), (c), (e) e (h) na Figura 15.2. O número de modos é denominado ***função partição***; é aproximadamente igual a $e^{\pi\sqrt{2n/3}} / 4n\sqrt{3}$. Essa quantidade é menor que 2^{n+1}, porém ainda muito maior do que qualquer polinômio em n. Todavia, não prosseguiremos nessa linha de raciocínio.

2 Isso não é um erro de ortografia. A palavra é realmente *memoização*, e não *memorização*. *Memoização* vem de *memo*, já que a técnica consiste em gravar um valor de modo que possamos consultá-lo mais tarde.

3 Usamos o termo “não ponderado” para distinguir esse problema do problema de encontrar caminhos mais curtos com arestas ponderadas, que veremos nos Capítulos 24 e 25. Podemos usar a técnica da busca em largura apresentada no Capítulo 22 para resolver o problema não ponderado.

4 Pode parecer estranho que programação dinâmica dependa de subproblemas que são ao mesmo tempo independentes e sobrepostos. Embora possam parecer contraditórios, esses requisitos descrevem duas noções diferentes, em vez de dois pontos no mesmo eixo. Dois subproblemas do mesmo subproblema são independentes se não compartilharem recursos. Dois subproblemas são sobrepostos se realmente forem o mesmo subproblema que ocorre como um subproblema de problemas diferentes.

5 Essa abordagem pressupõe que conhecemos o conjunto de todos os parâmetros de subproblemas possíveis e que estabelecemos a relação entre posições de tabela e subproblemas. Uma outra abordagem, mais geral, é memoizar usando hashing com os parâmetros do subproblema como chave.

6 Se o assunto do texto fosse arquitetura de castelos, talvez quiséssemos que *machicolation* aparecesse perto da raiz.

7 Sim, *machicolation* tem uma contraparte em francês: *mâchicoulis*.

8 Embora haja nove posições em um time de beisebol, N não é necessariamente igual a 9 porque o modo como alguns gerentes gerais pensam sobre posições é peculiar. Por exemplo, um gerente geral poderia considerar que há duas “posições” distintas para arremessadores (*pitchers*), isto é, os arremessadores destros e os arremessadores canhotos, além do primeiro arremessador da partida, dos arremessadores de longo prazo, que podem arremessar por vários turnos, e dos de curto prazo, que normalmente arremessam no máximo um turno.

9 ***Sabermétrica*** é a aplicação de análise estatística a registros de dados de jogos de beisebol. A sabermétrica dá vários modos para comparar valores relativos de jogadores individuais.

16 ALGORITMOS GULOSOS

Algoritmos para problemas de otimização, normalmente passam por uma sequência de etapas e cada etapa tem um conjunto de escolhas. Para muitos problemas de otimização é um exagero utilizar programação dinâmica para determinar as melhores escolhas: algoritmos mais simples e mais eficientes servirão. Um ***algoritmo guloso*** sempre faz a escolha que parece ser a melhor no momento em questão. Isto é, faz uma escolha localmente ótima, na esperança de que essa escolha leve a uma solução globalmente ótima. Este capítulo explora problemas de otimização para os quais os algoritmos gulosos dão soluções ótimas. Antes de ler este capítulo, você deve ler sobre programação dinâmica no Capítulo 15, em particular a Seção 15.3.

Algoritmos gulosos nem sempre produzem soluções ótimas, mas as produzem para muitos problemas. Primeiro, examinaremos na Seção 16.1 um problema simples, mas não trivial: o problema da seleção de atividades, para o qual um algoritmo guloso calcula eficientemente uma solução. Chegaremos ao algoritmo guloso, considerando primeiro uma abordagem de programação dinâmica e depois mostrando que sempre podemos fazer escolhas gulosas para chegar a uma solução ótima. A Seção 16.2 revê os elementos básicos da abordagem gulosa, dando uma abordagem direta para provar a correção de algoritmos gulosos. A Seção 16.3 apresenta uma aplicação importante das técnicas gulosas: o projeto de códigos de compressão de dados (Huffman). Na Seção 16.4, investigamos um pouco da teoria subjacente às estruturas combinatórias denominadas "matroides", para as quais um algoritmo guloso sempre produz uma solução ótima. Finalmente, a Seção 16.5 aplica matroides para resolver um problema de programação de tarefas de tempo unitário com prazos finais e multas.

O método guloso é bastante poderoso e funciona bem para uma ampla faixa de problemas. Capítulos posteriores apresentarão muitos algoritmos que podem ser vistos como aplicações do método guloso, entre eles os algoritmos de árvore geradora mínima (Capítulo 23), o algoritmo de Dijkstra para caminhos mais curtos que partem de uma origem única (Capítulo 24) e a heurística gulosa de cobertura de conjuntos de Chvátal (Capítulo 35). Os algoritmos de árvores geradoras mínimas são um exemplo clássico do método guloso. Embora este capítulo e o Capítulo 23 possam ser lidos independentemente um do outro, aconselhamos que você os leia juntos.

16.1 UM PROBLEMA DE SELEÇÃO DE ATIVIDADES

Nosso primeiro exemplo é o problema de programar várias atividades concorrentes que requerem o uso exclusivo de um recurso comum, com o objetivo de selecionar um conjunto de tamanho máximo de atividades mutuamente compatíveis. Suponha que tenhamos um conjunto $S = \{a_1, a_2, ..., a_n\}$ de n ***atividades*** propostas que desejam usar um recurso (por exemplo, uma sala de conferências) que só pode ser utilizado por uma única atividade por vez. Cada atividade a_i tem um ***tempo de início*** s_i e um ***tempo de término*** f_i , onde $0 \leq s_i < f_i < \infty$. Se selecionada, a atividade a_i ocorre durante o intervalo de tempo meio aberto $[s_i, f_i)$. As atividades a_i e a_j são ***compatíveis*** se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ não se sobrepõem. Isto é, a_i e a_j são compatíveis se $s_i \geq f_j$ ou $s_j \geq f_i$). No ***problema de seleção de atividades***, queremos selecionar um subconjunto de tamanho máximo de atividades mutuamente compatíveis. Supomos que as atividades estão organizadas em ordem monotonicamente crescente de tempo de término:

$$f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_{n-1} \leq f_n \tag{16.1}$$

(Veremos mais tarde como isso é vantajoso.) Por exemplo, considere o seguinte conjunto S de atividades:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Para este exemplo, o subconjunto $\{a_3, a_9, a_{11}\}$ consiste em atividades mutuamente compatíveis. Porém, não é um subconjunto máximo, já que o subconjunto $\{a_1, a_4, a_8, a_{11}\}$ é maior. De fato, $a_1, a_4, a_8, a_{11}\}$ é um dos subconjuntos maiores de atividades mutuamente compatíveis; um outro subconjunto maior é $\{a_2, a_4, a_9, a_{11}\}$.

Resolveremos esse problema em várias etapas. Começamos examinando uma solução de programação dinâmica na qual consideraremos várias escolhas para determinar quais subproblemas usar em uma solução ótima. Então, observaremos que precisamos considerar somente uma escolha — a escolha gulosa — e que, quando optamos pela escolha gulosa, restará apenas um subproblema. Com base nessas observações, desenvolveremos um algoritmo guloso recursivo para resolver o problema da programação de atividades. Concluiremos o processo de desenvolver uma solução gulosa, convertendo o algoritmo recursivo em um algoritmo iterativo. Embora as etapas que percorreremos nesta seção sejam ligeiramente mais complicadas do que é comum no desenvolvimento de um algoritmo guloso, elas ilustram a relação entre algoritmos gulosos e programação dinâmica.

A subestrutura ótima do problema de seleção de atividades

É fácil verificar que o problema de seleção de atividades exibe subestrutura ótima. Vamos denotar por S_{ij} o conjunto de atividades que começam após o término da atividade a_i e terminam antes do início da atividade a_j. Suponha que queremos determinar um conjunto máximo de atividades mutuamente compatíveis em S_{ij}, e suponha ainda mais que tal subconjunto máximo é A_{ij}, que inclui alguma atividade a_k. Incluindo a_k em uma solução ótima, ficamos com dois subproblemas em mãos: determinar atividades mutuamente compatíveis no conjunto S_{ik} (atividades que começam após o término da atividade a_i e terminam antes do início da atividade a_k) e determinar atividades mutuamente compatíveis no conjunto S_{kj} (atividades que começam após o término da atividade a_k e terminam antes do início da atividade a_j). Sejam $A_{ik} = A_{ij} \cap S_{ik}$ e $A_{kj} = A_{ij} \cap S_{kj}$, de modo que A_{ik} contém as atividades em A_{ij} que terminam antes do início de a_k e A_{kj} contém as atividades em A_{ij} que começam após o término de a_k. Assim, temos $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ e, portanto, o conjunto de tamanho máximo A_{ij} de atividades mutuamente compatíveis em S_{ij} consiste em $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ atividades.

O argumento usual de recortar e colar mostra que a solução ótima A_{ij} deve incluir também soluções ótimas para os dois subproblemas para S_{ik} e S_{kj}. Se pudéssemos determinar um conjunto A'_{kj} de atividades mutuamente compatíveis em S_{kj} onde $|A'_{kj}| > |A'_{kj}|$, poderíamos usar A'_{kj} , m vez de A_{kj}, em uma solução para os subproblemas para S_{ij}. Poderíamos ter construído um conjunto de $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ atividades mutuamente compatíveis, o que contradiz a suposição de que A_{ij} seja uma solução ótima. Um argumento simétrico se aplica às atividades em S_{ik}.

Esse modo de caracterizar subestrutura ótima sugere que poderíamos resolver o problema de seleção de atividades por programação dinâmica. Se denotarmos o tamanho de uma solução ótima para o conjunto S_{ij} por $c[i, j]$ teremos a recorrência

$$c[i,j] = c[i,k] + c[k,j] + 1.$$

É claro que, se não soubermos que uma solução ótima para o conjunto S_{ij} inclui a atividade a_k, teremos de examinar todas as atividades em S_{ij} para determinar qual delas escolher, de modo que

$$c[i,j] = \begin{cases} 0 & se\ S_{ij} = \varnothing \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & se\ S_{ij} \neq \varnothing \end{cases} \qquad (16.2)$$

Então, poderemos desenvolver um algoritmo recursivo e memoizá-lo, ou trabalhar de baixo para cima e preencher entradas de tabela durante o processo. Porém, estaríamos ignorando uma outra característica importante do problema de seleção de atividades que podemos usar e que seria muito vantajoso.

Fazendo a escolha gulosa

E se pudéssemos escolher uma atividade para acrescentar à nossa solução ótima sem ter de resolver primeiro todos os subproblemas? Isso nos pouparia de ter de considerar todas as escolhas inerentes à recorrência (16.2). Na verdade, no caso do problema de seleção de atividades precisamos considerar somente uma escolha: a escolha gulosa.

O que quer dizer escolha gulosa para o problema de seleção de atividades? A intuição sugere que deveríamos escolher uma atividade que deixa o recurso disponível para o maior número possível de outras atividades. Agora, entre as atividades que acabamos escolhendo, uma deve ser a primeira a terminar. Portanto, nossa intuição nos diz para escolher a atividade em S que tenha o tempo de término mais cedo, já que isso deixaria o recurso disponível para o maior número possível de atividades que ocorram depois dessa. (Se mais de uma atividade em S tiver o mesmo tempo de término mais cedo, poderemos escolher qualquer uma delas.) Em outras palavras, visto que as atividades são ordenadas em ordem monotônica crescente por tempo de término, a escolha gulosa é a atividade a_1. Escolher a primeira atividade a terminar não é o único modo de fazer uma escolha gulosa para esse problema; o Exercício 16.1-3 pede que você explore outras possibilidades.

Se fizermos a escolha gulosa, restará somente um subproblema para resolver: determinar atividades que começam após o término de a_1. Por que não temos de considerar atividades que terminam antes de a[1] começar? Temos que $s_1 < f_1$, e f_1 é o tempo mais cedo de término de qualquer atividade; portanto, nenhuma atividade pode ter um tempo de término menor ou igual a s_1. Assim, todas as atividades que são compatíveis com a atividade a_1 devem começar depois que a[1] terminar.

Além do mais, já estabelecemos que o problema de seleção de atividades exibe subestrutura ótima. Seja $S_k = \{a_i \in S : s_i \geq f_k\}$ o conjunto de atividades que começam após o término de a[k]. Se fizermos a escolha gulosa da atividade a , S permanecerá como o único problema a resolver.[1] A subestrutura ótima nos diz que, se a_1 estiver na solução ótima, uma solução ótima para o problema original consistirá na atividade a_1 e todas as atividades em uma solução ótima para o subproblema S_1.

Resta uma grande pergunta: nossa intuição está correta? A escolha gulosa — na qual escolhemos a primeira atividade a terminar — é sempre parte de alguma solução ótima? O teorema que apresentamos a seguir, prova que é.

Teorema 16.1

Considere um subproblema qualquer não vazio S_k, e seja a_m uma atividade em S_k com o tempo de término mais cedo. Então, a_m estará incluída em algum subconjunto de tamanho máximo de atividades mutuamente compatíveis de S_k.

Prova Seja A_k um subconjunto de tamanho máximo de atividades mutuamente compatíveis em S_k, e seja a_j a atividade em A_k que tem o tempo de término mais cedo. Se $a_j = a_m$, terminamos aqui, visto que já mostramos que a_m está em algum subconjunto de tamanho máximo de atividades mutuamente compatíveis de S_k. Se $a_j \neq a_m$, considere o conjunto $A_k' = A_k - \{a_j\} \cup \{a_m\}$, que é A_k substituindo a_j por a_m. As atividades em A_k' são disjuntas, o que decorre porque as atividades em A_k são disjuntas, a_j é a primeira atividade a terminar em A_k e $f_m \leq f_j$. Visto que $|A_k'| = |A_k|$, concluímos que A_k' é um subconjunto de tamanho máximo de atividades mutuamente compatíveis de S_k e ele inclui a_m.

Assim, vemos que, ainda que pudéssemos resolver o problema da seleção de atividades com programação dinâmica, não precisamos fazê-lo. (Ademais, ainda não verificamos se o problema de seleção de atividades tem subproblemas sobrepostos.)

Em vez disso, podemos escolher repetidamente a atividade que termina primeiro, manter somente as atividades compatíveis com essa atividade e repetir o processo até não restar nenhuma atividade. Além do mais, como sempre escolhemos a atividade que tem o tempo de término mais cedo, os tempos de término das atividades que escolhermos deve crescer estritamente. Podemos considerar cada atividade apenas uma vez no total, em ordem monotonicamente crescente de tempos de término.

Um algoritmo para resolver o problema de seleção de atividades não precisa funcionar de baixo para cima, como um algoritmo de programação dinâmica baseado em tabela. Em vez disso, pode trabalhar de cima para baixo escolhendo uma atividade para colocar na solução ótima e resolvendo o problema de escolher atividades entre as que são compatíveis com as já escolhidas. Algoritmos gulosos, normalmente têm o seguinte projeto de cima para baixo: faça um escolha e resolva um subproblema, em vez da técnica de baixo para cima, que resolve subproblemas antes de fazer uma escolha.

Um algoritmo guloso recursivo

Agora, que já vimos como evitar a abordagem da programação dinâmica e usar um algoritmo guloso de cima para baixo no lugar dela, podemos escrever um procedimento recursivo direto para resolver o problema da seleção de atividades. O procedimento RECURSIVE-ACTIVITY-SELECTOR toma os tempos de início e término das atividades, representados como arranjos s e f,[2] o índice k que define o subproblema S_k que ele deve resolver e o tamanho n do problema original, e retorna um conjunto de tamanho máximo de atividades mutuamente compatíveis em S_k. Consideramos que as n atividades de entrada já estão organizadas em ordem monotonicamente crescente de tempos de término de acordo com a equação (16.1). Se não, podemos ordená-las nessa ordem no tempo $O(n \lg n)$, rompendo vínculos arbitrariamente. Para começar, acrescentamos a atividade fictícia a_0 com $f_0 = 0$, de modo que o subproblema S_0 é o conjunto inteiro de atividades S. A chamada inicial, que resolve o problema inteiro, é RECURSIVE-ACTIVITY-SELECTOR.$(s, f, 0, n)$.

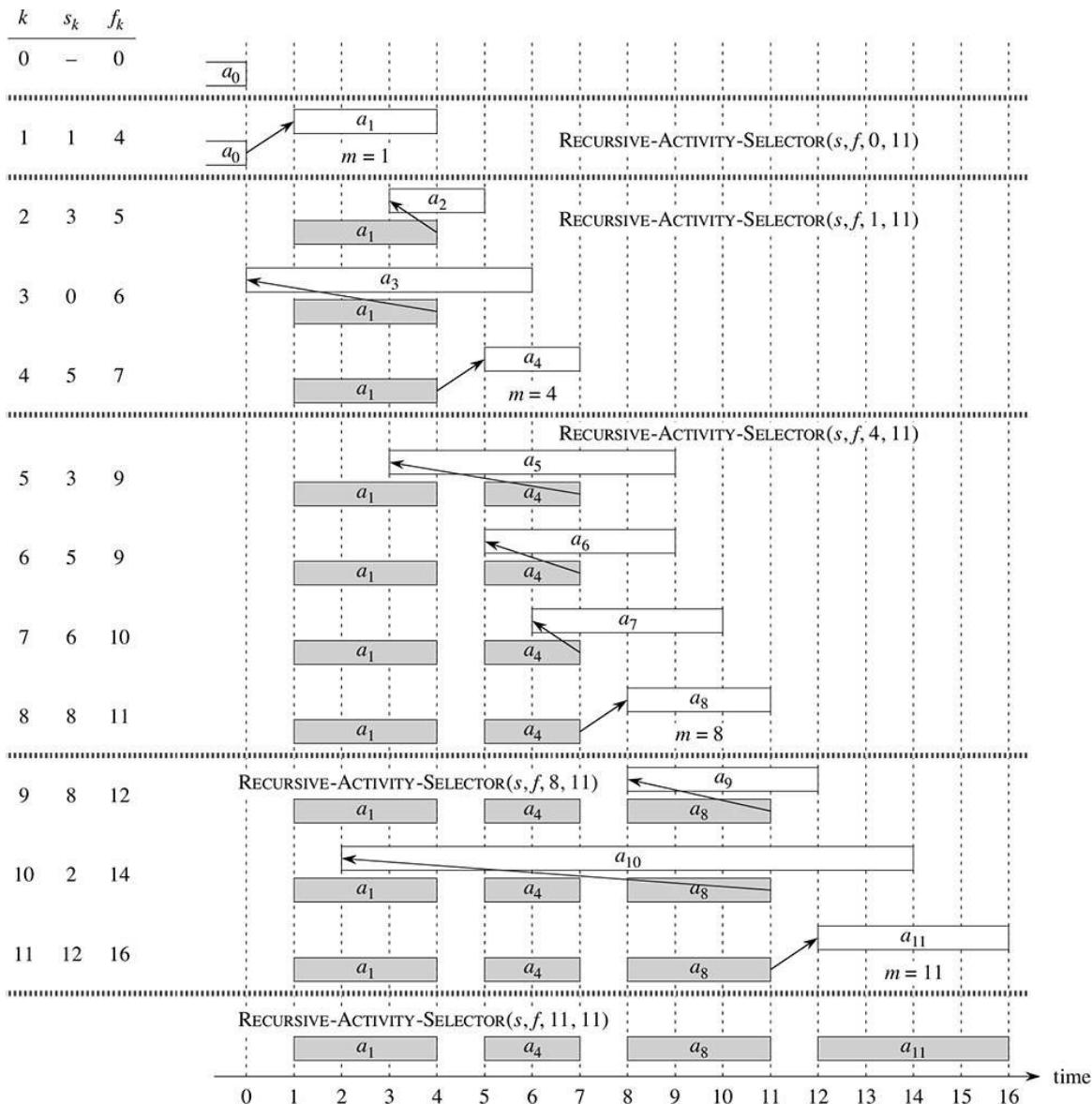
RECURSIVE-ACTIVITY-SELECTOR (s, f, k, n)
1 $m = k + 1$
2 **while** $m \le n$ e $s[m] < f[k]$ // encontrar em Sk a primeira atividade que termina
3 $\quad m = m + 1$
4 **if** $m \le n$
5 $\quad$ **return** $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR. (s, f, m, n)
6 **else return** $\emptyset$

A Figura 16.1 mostra como o algoritmo funciona. Em determinada chamada recursiva RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n), o laço **while** das linhas 2–3 procura em S_k a primeira atividade que termina. O laço examina $a_k + 1$, $a_k + 2$,..., a_n, até encontrar a primeira atividade a_m que é compatível com a_k ; tal atividade tem $s_m \ge f_k$. Se o laço terminar porque encontrou tal atividade, a linha 5 retorna a união de $\{a_m\}$ com o subconjunto de tamanho máximo de S_m retornado pela chamada recursiva RECURSIVE-ACTIVITY-SELECTOR (s, f, m, n). Alternativamente, o laço pode terminar porque $m > n$, caso em que já examinamos todas as atividades em S_k e não encontramos uma que seja compatível com a_k. Nesse caso, $S_k =$ 0/ e, portanto, o procedimento devolve $\emptyset$ na linha 6.

Considerando que as atividades já foram ordenadas por tempos de término, o tempo de execução da chamada RECURSIVE-ACTIVITY-SELECTOR $(s, f, 0, n)$ é $\Theta(n)$ o que podemos verificar da maneira descrita a seguir. Em todas as chamadas recursivas, cada atividade é examinada exatamente uma vez no teste do laço **while** da linha 2. Em particular, a atividade a_i é examinada na última chamada feita na qual $k < i$.

Um algoritmo guloso iterativo

É fácil converter nosso procedimento recursivo em um iterativo. O procedimento RECURSIVE-ACTIVITY-SELECTOR é quase "recursivo de cauda" (veja o Problema 7-4): ele termina com uma chamada recursiva a si mesmo seguida por uma operação de união. Em geral, transformar um procedimento recursivo de cauda em uma forma iterativa é uma tarefa direta; de fato, alguns compiladores para certas linguagens de programação executam essa tarefa automaticamente. Como está escrito, RECURSIVE-ACTIVITY-SELECTOR funciona para subproblemas S_k , isto é, subproblemas que consistem nas últimas atividades a terminar.

O procedimento GREEDY-ACTIVITY-SELECTOR é uma versão iterativa do procedimento RECURSIVE-ACTIVITY-SELECTOR. Ele também considera que as atividades de entrada estão organizadas em ordem monotonicamente crescente de tempos de término. O procedimento reúne atividades selecionadas em um conjunto A e retorna esse conjunto quando termina.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1   n = s.comprimento
2   A = {a_1}
3   k = 1
4   for m = 2 to n
5       if s[m] ≥ f[k]
6           A = A ∪ {a_m}
7           k = m
8   return A
```

O procedimento funciona da maneira descrita a seguir. A variável k indexa a adição mais recente a A, correspondente à atividade a_k na versão recursiva. Visto que consideramos as atividades em ordem monotonicamente crescente de tempos de término, f_k é sempre o tempo de término máximo de qualquer atividade em A. Isto é,

$$f_k = \max\{f_i : a_i \in A\}. \quad (16.3)$$

As linhas 2–3 selecionam a atividade a_1, inicializam A para conter apenas essa atividade e inicializam k para indexar essa atividade. O laço **for** das linhas 4–7 encontra a atividade que termina mais cedo em S_k. O laço considera cada atividade a_m por vez e acrescenta a_m a A se ela for compatível com todas as atividades selecionadas anteriormente; tal atividade é a que termina mais cedo S_k. Para ver se a atividade a_m é compatível com cada atividade presente em A no momento em questão, basta utilizar a equação (16.3) para verificar (na linha 5) se seu tempo de início s_m não é anterior ao tempo de término f_k da atividade mais recentemente adicionada a A. Se a atividade a_m é compatível, as linhas 6–7 adicionam a atividade a_m a A e atribuem k com m. O conjunto A retornado pela chamada GREEDY-ACTIVITY-SELECTOR(s, f) é exatamente o conjunto retornado pela chamada RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Figura 16.1 O funcionamento de RecuRsive-Activity-selectoR para as 11 atividades dadas anteriormente. As atividades consideradas em cada chamada recursiva aparecem entre linhas horizontais. A atividade fictícia a_0 termina no tempo 0 e a chamada inicial, RecuRsive-Activity-selectoR(s, f, 0, 11) seleciona a atividade a_1 . Em cada chamada recursiva, as atividades que já foram selecionadas estão sombreadas, e a atividade mostrada em branco está sendo considerada. Se o tempo de início de uma atividade ocorre antes do tempo de término da atividade mais recentemente adicionada (a seta entre elas aponta para a esquerda), ela é rejeitada. Caso contrário (a seta aponta diretamente para cima ou para a direita), ela é selecionada. A última chamada recursiva, RecuRsive-Activity-selectoR(s, f, 11, 11), devolve 0/. O conjunto resultante de atividades selecionadas é $\{a_1, a_4, a_8, a_{11}\}$.

Como a versão recursiva, o procedimento Greedy-Activity-Selector programa um conjunto de n atividades no tempo $Q(n)$, considerando que as atividades já estavam ordenadas inicialmente por seus tempos de término.

Exercícios

16.1-1 Dê um algoritmo de programação dinâmica para o problema de seleção de atividades, baseado na recorrência (16.2). Seu algoritmo deve calcular os tamanhos $c[i, j]$ como definidos anteriormente e também produzir o subconjunto de tamanho máximo de atividades mutuamente compatíveis. Suponha que as entradas estão ordenadas como na equação (16.1). Compare o tempo de execução da sua solução com o tempo de

execução de GREEDY-ACTIVITY-SELECTOR.

16.1-2 Suponha que, em vez de sempre selecionar a primeira atividade a terminar, selecionemos a última atividade a começar que seja compatível com todas as atividades selecionadas anteriormente. Descreva como essa abordagem é um algoritmo guloso e prove que ela produz uma solução ótima.

16.1-3 Não é qualquer abordagem gulosa para o problema de seleção de atividades que produz um conjunto de tamanho máximo de atividades mutuamente compatíveis. Dê um exemplo para mostrar que a abordagem de selecionar a atividade de menor duração entre aquelas que são compatíveis com atividades selecionadas anteriormente não funciona. Faça o mesmo para a abordagem de sempre selecionar a atividade que se sobrepõe ao menor número de outras atividades e para a abordagem de sempre selecionar a atividade restante compatível com o tempo de início mais cedo.

16.1-4 Suponha que temos um conjunto de atividades para programar entre um grande número de salas de conferência. Desejamos programar todas as atividades usando o menor número possível de salas de conferências. Dê um algoritmo guloso eficiente para determinar qual atividade deve usar cada sala de conferências.

(Este problema também é conhecido como ***problema de colorir grafos de intervalos***. Podemos criar um grafo de intervalos cujos vértices são as atividades dadas e cujas arestas conectam atividades incompatíveis. O menor número de cores necessárias para colorir cada vértice de modo que dois vértices adjacentes nunca tenham a mesma cor corresponde a encontrar o menor número de salas de conferências necessárias para programar todas as atividades dadas.)

16.1-5 Considere uma modificação para o problema de seleção de atividades no qual, além de um tempo de início e fim, cada atividade a_i tem um valor v_i. O objetivo não é mais maximizar o número de atividades programadas, mas maximizar o valor total das atividades programadas. Isto é, queremos escolher um conjunto A de atividades compatíveis tal que $\sum_{a_k \in A} v_k$ seja maximizado. Dê um algoritmo de tempo polinomial para esse problema.

16.2 ELEMENTOS DA ESTRATÉGIA GULOSA

Um algoritmo guloso obtém uma solução ótima para um problema fazendo uma sequência de escolhas. Para cada ponto de decisão, o algoritmo escolhe a opção que parece melhor no momento. Essa estratégia heurística nem sempre produz uma solução ótima, mas, como vimos no problema de seleção de atividades, algumas vezes, funciona. Esta seção discute algumas propriedades gerais de métodos gulosos.

O processo que seguimos na Seção 16.1 para desenvolver um algoritmo guloso foi um pouco mais complicado que o normal. Seguimos estas etapas:

1. Determinar a subestrutura ótima do problema.
2. Desenvolver uma solução recursiva. (Para o problema de seleção de atividades, formulamos a recorrência (16.2), mas evitamos desenvolver um algoritmo recursivo baseado nessa recorrência.)
3. Provar que, se fizermos a escolha gulosa, restará somente um subproblema.
4. Provar que, é sempre seguro fazer a escolha gulosa. (As etapas 3 e 4 podem ocorrer em qualquer ordem.)
5. Desenvolver um algoritmo recursivo que implemente a estratégia gulosa.
6. Converter o algoritmo recursivo em um algoritmo iterativo.

Seguindo essas etapas, vimos bem detalhadamente os fundamentos da programação dinâmica de um algoritmo guloso. Por exemplo, no problema da seleção de atividades, primeiro definimos os subproblemas S_{ij}, no qual i e j variavam. Então, constatamos que, se sempre fizéssemos a escolha gulosa, poderíamos restringir os subproblemas à forma S_k.

Alternativamente, poderíamos ter conformado nossa subestrutura ótima tendo em mente uma escolha gulosa, de modo que a escolha deixasse apenas um problema para resolver. No problema de seleção de atividades, poderíamos ter começado descartando o segundo índice e definindo subproblemas da forma S_k. Então, poderíamos ter provado que uma escolha gulosa (a primeira atividade a_m a terminar em S_k), combinada com uma solução ótima para o conjunto restante S_m de atividades compatíveis, produz uma solução ótima para S_k. De modo mais geral, projetamos algoritmos gulosos de acordo com a seguinte sequência de etapas:

1. Expressar o problema de otimização como um problema no qual fazemos uma escolha e ficamos com um único subproblema para resolver.
2. Provar que sempre existe uma solução ótima para o problema original que usa a escolha gulosa, de modo que a escolha gulosa é sempre segura.
3. Demonstrar subestrutura ótima mostrando que, tendo feito a escolha gulosa, o que resta é um subproblema com a seguinte propriedade: se combinarmos uma solução ótima para o subproblema com a escolha gulosa que fizemos, chegamos a uma solução ótima para o problema original.

Usaremos esse processo mais direto em seções posteriores deste capítulo. Apesar disso, embaixo de todo algoritmo guloso, quase sempre existe uma solução de programação dinâmica mais incômoda.

Como saber se um algoritmo guloso resolverá determinado problema de otimização? Nenhum método funciona todas as vezes, mas a propriedade de escolha gulosa e a subestrutura ótima são os dois componentes fundamentais. Se pudermos demonstrar que o problema tem essas propriedades, estaremos no bom caminho para desenvolver um algoritmo guloso para ele.

Propriedade de escolha gulosa

O primeiro componente fundamental é a ***propriedade de escolha gulosa***: podemos montar uma solução globalmente ótima fazendo escolhas (gulosas) locais ótimas. Em outras palavras, quando estamos considerando qual escolha fazer, escolhemos a que parece melhor para o problema em questão, sem considerar resultados de subproblemas.

É nesse ponto que os algoritmos gulosos são diferentes da programação dinâmica. Na programação dinâmica, fazemos uma escolha em cada etapa, mas, normalmente a escolha depende das soluções para subproblemas. Consequentemente, em geral, resolvemos problemas de programação dinâmica de baixo para cima, passando de subproblemas menores para subproblemas maiores. (Alternativamente, podemos resolvê-los de cima para baixo, mas usando memoização. É claro que, mesmo que o código funcione de cima para baixo, ainda temos de resolver os subproblemas antes de fazer uma escolha.) Em um algoritmo guloso, fazemos qualquer escolha que pareça melhor no momento e depois resolvemos o subproblema que resta. A escolha feita por um algoritmo guloso pode depender das escolhas até o momento em questão, mas não pode depender de nenhuma escolha futura ou das soluções para subproblemas. Assim, diferentemente da programação dinâmica, que resolve os subproblemas antes de fazer a primeira escolha, um algoritmo guloso faz sua primeira escolha antes de resolver qualquer subproblema. Um algoritmo de programação dinâmica age de baixo para cima, ao passo que, uma estratégia gulosa em geral age de cima para baixo, fazendo uma escolha gulosa após outra, reduzindo cada instância do problema dado a uma instância menor.

É claro que, temos de provar que uma escolha gulosa em cada etapa produz uma solução globalmente ótima. Normalmente, como no caso do Teorema 16.1, a prova examina uma solução globalmente ótima para algum subproblema. Então, mostra como modificar a solução para usar a escolha gulosa no lugar de alguma outra escolha, resultando em um subproblema semelhante, porém menor.

Normalmente, podemos fazer a escolha gulosa com maior eficiência do que quando temos de considerar um conjunto de escolha mais amplo. Por exemplo, no problema de seleção de atividades, considerando que já tivéssemos organizado as atividades em ordem monotonicamente crescente de tempos de término, precisávamos examinar cada atividade apenas uma vez. Reprocessando a entrada ou usando uma estrutura de dados apropriada (quase sempre uma fila de prioridades), muitas vezes, podemos fazer escolhas gulosas rapidamente, produzindo assim um algoritmo eficiente.

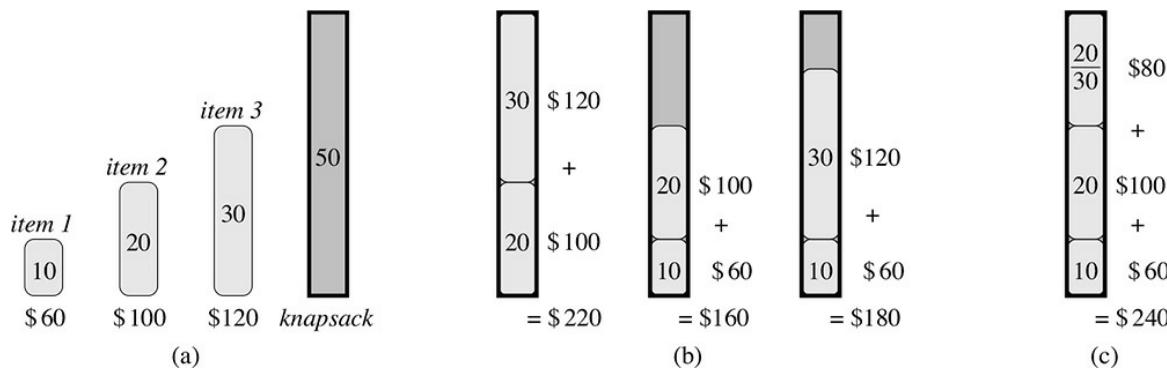
Subestrutura ótima

Um problema exibe ***subestrutura ótima*** se uma solução ótima para o problema contiver soluções ótimas para subproblemas. Essa propriedade é um componente fundamental para avaliar a aplicabilidade da aplicação da programação dinâmica e também a de algoritmos gulosos. Como exemplo de subestrutura ótima, lembre-se de como demonstramos na Seção 16.1 que, se uma solução ótima para o subproblema S_{ij} incluir uma atividade a_k, então ela também deve conter soluções ótimas para os subproblemas S_{ik} e S_{kj}. Dada essa subestrutura ótima, demonstramos que, se soubéssemos qual atividade usar como a_k, poderemos construir uma solução ótima para S_{ij} selecionando a_k juntamente com todas as atividades em soluções ótimas para os subproblemas S_{ik} e S_{kj}. Com base nessa observação de subestrutura ótima, pudemos criar a recorrência (16.2) que descrevia o valor de uma solução ótima.

Normalmente, usamos uma abordagem mais direta em relação à subestrutura ótima quando a aplicamos a algoritmos gulosos. Conforme já mencionamos, podemos nos permitir o luxo de supor que chegamos a um subproblema por termos feito a escolha gulosa no problema original. Na realidade, basta que demonstremos que uma solução ótima para o subproblema, combinada com a escolha gulosa já feita, produz uma solução ótima para o problema original. Esse esquema utiliza implicitamente indução em relação aos subproblemas para provar que fazer a escolha gulosa em cada etapa produz uma solução ótima.

Estratégia gulosa versus programação dinâmica

Como as estratégias gulosa e de programação dinâmica exploram subestrutura ótima, bem que você poderia ser tentado a gerar uma solução de programação dinâmica para um problema quando uma solução gulosa seria suficiente ou, ao contrário, achar erroneamente que uma solução gulosa funciona quando, na verdade, seria preciso uma solução de programação dinâmica. Para ilustrar as sutilezas entre as duas técnicas, vamos investigar duas variantes de um problema clássico de otimização.

Apresentamos a seguir, o ***problema da mochila 0-1***. Um ladrão que assalta uma loja encontra n itens. O i-ésimo item vale v_i reais e pesa w_i quilos, onde v_i e w_i são inteiros. Ele deseja levar consigo a carga mais valiosa possível, mas só pode carregar no máximo W quilos em sua mochila, sendo W um número inteiro. Que itens ele deve levar? (Esse problema da mochila chama-se 0-1 porque, para cada item, o ladrão tem de decidir se o carrega consigo ou se o deixa para trás; ele não pode levar uma quantidade fracionária de um item nem um item mais de uma vez.)

No ***problema fracionário da mochila***, a configuração é a mesma, mas o ladrão pode levar frações de itens, em vez de ter de fazer uma escolha binária (0-1) para cada item. Para perceber melhor a diferença, imagine que um item no problema da mochila 0-1 seja como um lingote de ouro, enquanto um item no problema da mochila fracionário seria semelhante a ouro em pó.

Ambos os problemas da mochila exibem a propriedade de subestrutura ótima. No caso do problema 0-1, considere a carga mais valiosa, que pesa no máximo W quilos. Se removermos o item j dessa carga, a carga restante deverá ser a carga mais valiosa que pese no máximo $W - w_j$ que o ladrão pode levar dos $n - 1$ itens originais, excluindo j. Por comparação, no caso do problema fracionário, considere que, se removermos um peso w de um item j da carga ótima, a carga restante deverá ser a carga mais valiosa que pese no máximo $W - w$ que o ladrão pode levar dos $n - 1$ itens originais mais $w_j - w$ quilos do item j.

Embora os problemas sejam semelhantes, podemos resolver o problema fracionário da mochila por uma estratégia gulosa, mas não podemos usar essa mesma estratégia para resolver o problema 0-1. Para resolver o problema fracionário, primeiro calculamos o valor por quilo v_i/w_i para cada item. Obedecendo a uma estratégia gulosa, o ladrão começa levando o máximo possível do item que tenha o maior valor por quilo. Se o suprimento desse item se esgotar e o ladrão ainda puder carregar mais, levará o máximo possível do próximo item que tenha o maior valor por quilo e assim por diante até alcançar seu peso limite W. Portanto, ordenando os itens por valor por quilo, o algoritmo guloso é executado no tempo $O(n \lg n)$. Deixamos para o Exercício 16.2-1 a prova de que o problema fracionário da mochila tem a propriedade de escolha gulosa.

Para ver que essa estratégia gulosa não funciona para o problema da mochila 0-1, considere a instância do problema ilustrada na Figura 16.2(a). Esse exemplo tem três itens e uma mochila que pode conter 50 quilos. O item 1 pesa 10 quilos e vale 60 reais. O item 2 pesa 20 quilos e vale 100 reais. O item 3 pesa 30 quilos e vale 120 reais. Portanto, o valor por quilo do item 1 é 6 reais por quilo, que é maior que o valor por quilo do item 2 (5 reais por quilo) ou do item 3 (4 reais por quilo). Assim, a estratégia gulosa levaria o item 1 primeiro. Porém, como podemos ver pela análise de caso na Figura 16.2(b), a solução ótima leva os itens 2 e 3 e deixa o item 1 para trás. As duas soluções possíveis que envolvem o item 1 são subótimas.

Contudo, por comparação, a estratégia gulosa para o problema fracionário, que leva o item 1 primeiro, realmente produz uma solução ótima, como mostra a Figura 16.2(c). Levar o item 1 não funciona no problema 0-1 porque o ladrão não consegue encher sua mochila até a capacidade máxima, e o espaço vazio reduz o valor efetivo por quilo de sua carga. No problema 0-1, quando consideramos incluir um item na mochila, temos de comparar a solução para o subproblema que inclua tal item com a solução para o subproblema que exclua esse mesmo item, antes de podermos fazer a escolha. O problema formulado desse modo dá origem a muitos subproblemas sobrepostos — uma marca registrada da programação dinâmica. Realmente, como o Exercício 16.2-2 pede que você mostre, podemos usar programação dinâmica para resolver o problema 0-1.

Figura 16.2 Um exemplo que mostra que a estratégia gulosa não funciona para o problema da mochila 0-1. (**a**) O ladrão deve selecionar um subconjunto dos três itens mostrados cujo peso não pode exceder 50 quilos. (**b**) O subconjunto ótimo inclui os itens 2 e 3. Qualquer solução com o item 1 é subótima, embora o item 1 tenha o maior valor por quilo. (**c**) Para o problema fracionário da mochila, tomar os itens em ordem de maior valor por quilo produz uma solução ótima.

Exercícios

16.2-1 Prove que o problema fracionário da mochila tem a propriedade de escolha gulosa.

16.2-2 Dê uma solução de programação dinâmica para o problema da mochila 0-1 que seja executado no tempo $O(nW)$, onde n é número de itens e W é o peso máximo de itens que o ladrão pode pôr em sua mochila.

16.2-3 Suponha que, em um problema da mochila 0-1, a ordem dos itens quando ordenados por peso crescente seja igual à ordem quando ordenados por valor decrescente. Dê um algoritmo eficiente para determinar uma

solução ótima para essa variante do problema da mochila e mostre que seu algoritmo é correto.

16.2-4 O professor Gekko sempre sonhou em atravessar o estado de Dakota do Norte de patins. Ele planeja atravessar o estado pela rodovia U.S. 2, que vai de Grand Forks, na divisa leste com o estado de Minnesota, até Williston, perto da divisa oeste com o estado de Montana. O professor pode carregar dois litros de água e patinar m quilômetros antes de esgotar seu estoque de água. (Como o relevo de Dakota do Norte é relativamente plano, ele não precisa se preocupar com beber uma quantidade maior de água em trechos em aclive do que em trechos em declive e planos.) O professor partirá de Grand Forks com dois litros de água completos. O mapa oficial do estado de Dakota do Norte mostra todos os lugares ao longo da rodovia U.S. 2, onde ele poderá reabastecer seu estoque de água e as distâncias entre eles.

A meta do professor é minimizar o número de paradas para reabastecimento de água ao longo de sua rota pelo estado. Dê um método eficiente pelo qual ele possa determinar em quais lugares deverá repor seu estoque de água. Prove que tal estratégia produz uma solução ótima e dê o tempo de execução dessa estratégia.

16.2-5 Descreva um algoritmo eficiente que, dado um conjunto $\{x_1, x_2, ..., x_n\}$ de pontos na reta de números reais, determina o menor conjunto de intervalos fechados de comprimento unitário que contém todos os pontos dados. Mostre que seu algoritmo é correto.

16.2-6 ★ Mostre como resolver o problema fracionário da mochila no tempo $O(n)$.

16.2-7 Suponha que você tenha dois conjuntos A e B, cada um contendo n inteiros positivos, e que possa reordenar cada conjunto como preferir. Depois da reordenação, seja a_i o i-ésimo elemento do conjunto A e seja b_i o i-ésimo elemento do conjunto B. Então, você obtém uma compensação de $\prod_{i=1}^{n} a_i^{b_i}$. Dê um algoritmo que maximize essa compensação. Prove que seu algoritmo maximiza a compensação e dê seu tempo de execução.

16.3 Códigos de Huffman

Códigos de Huffman comprimem dados muito efetivamente: economias de 20-90% são típicas, dependendo das características dos dados que estão sendo comprimidos. Consideramos os dados como uma sequência de caracteres. O algoritmo guloso de Huffman utiliza uma tabela que dá o número de vezes que cada caractere ocorre (isto é, suas frequências) para elaborar um modo ótimo de representar cada caractere como uma cadeia binária.

Suponha que tenhamos um arquivo de dados de 100.000 caracteres que desejamos armazenar compactamente. Observamos que os caracteres no arquivo ocorrem com as frequências dadas pela Figura 16.3. Isto é, somente seis caracteres diferentes aparecem, e o caractere a ocorre 45.000 vezes.

Há muitas opções para representar tal arquivo de informações. Aqui, consideramos o problema de projetar um ***código de caracteres binários*** (ou ***código***, para abreviar) no qual cada caractere seja representado por uma cadeia binária única que denominaremos **palavra de código**. Se usarmos um ***código de comprimento fixo***, precisaremos de três bits para representar seis caracteres: a = 000, b = 001, ..., f = 101. Esse método requer 300.000 bits para codificar o arquivo inteiro. Podemos fazer algo melhor?

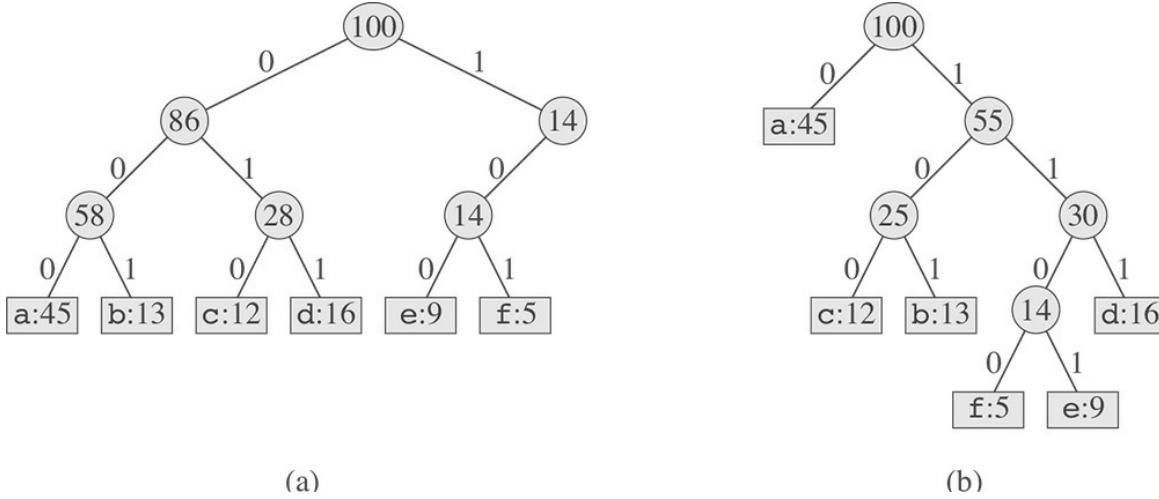
Um ***código de comprimento variável*** pode funcionar consideravelmente melhor que um código de comprimento fixo atribuindo palavras de código curtas a caracteres frequentes e palavras de código longas a caracteres pouco frequentes. A Figura 16.3 mostra um código desse tipo; aqui, a cadeia de 1 bit 0 representa a, e a cadeia de 4 bits 1100 representa f. Esse código requer

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1.000 = 224.000 \text{ bits}$$

para representar o arquivo, uma economia de aproximadamente 25%. De fato, esse é um código de caracteres ótimo para esse arquivo, como veremos.

Códigos de prefixo

Consideramos aqui apenas códigos nos quais nenhuma palavra de código seja também um prefixo de alguma outra palavra de código. Tais códigos são denominados ***códigos de prefixo***.[3] Embora não o provemos aqui, um código de prefixo sempre consegue a compressão de dados ótima em qualquer código de caracteres, portanto não haverá prejuízo para a generalidade se restringirmos nossa atenção a códigos de prefixo.

Codificar é sempre simples para qualquer código de caracteres binários; simplesmente concatenamos as palavras de código que representam cada caractere do arquivo. Por exemplo, com o código de prefixo de comprimento variável da Figura 16.3, codificamos o arquivo de três caracteres abc como 0 · 101 · 100 = 0101100, onde "·" denota concatenação.

Códigos de prefixo são desejáveis porque simplificam a decodificação. Como nenhuma palavra de código é um prefixo de qualquer outra, a palavra de código que inicia um arquivo codificado não é ambígua. Podemos simplesmente identificar a palavra de código inicial, traduzi-la de volta para o caractere original e repetir o processo de decodificação no restante do arquivo codificado. Em nosso exemplo, a cadeia 001011101 é analisada unicamente como 0 · 0 · 101 · 1101, que é decodificada como aabe.

O processo de decodificação precisa de uma representação conveniente para o código de prefixo, de modo que possamos extrair facilmente a palavra de código inicial. Uma árvore binária cujas folhas são os caracteres nos dá tal representação. Interpretamos a palavra de código binária para um caractere como o caminho simples da raiz até esse caractere, onde 0 significa "vá para o filho à esquerda" e 1 significa "vá para o filho à direita". A Figura 16.4 mostra as árvores para os dois códigos do nosso exemplo. Observe que elas não são árvores de busca binária, já que as folhas não precisam aparecer em sequência ordenada e os nós internos não contêm chaves de caracteres.

Um código ótimo para um arquivo é sempre representado por uma árvore binária *cheia*, na qual cada nó que não é uma folha tem dois filhos (veja o Exercício 16.3-2). O código de comprimento fixo em nosso exemplo não é ótimo, já que sua árvore, mostrada na Figura 16.4(a), não é uma árvore binária cheia: ela contém palavras de código que começam com 10..., mas nenhuma com 11.... Visto que agora podemos restringir nossa atenção a árvores binárias completas, podemos dizer que, se C é o alfabeto do qual os caracteres são extraídos e todas as frequências de caracteres são positivas, então a árvore para um código de prefixo ótimo tem exatamente $|C|$ folhas, uma para cada letra do alfabeto, e exatamente $|C| - 1$ nós internos (veja o Exercício B.5-3).

	a	b	c	d	e	f
Frequência (em milhares)	45	13	12	16	9	5
Palavra de código de comprimento fixo	000	001	010	011	100	101
Palavra de código de comprimento variável	0	101	100	111	1101	1100

Figura 16.3 Um problema de codificação de caracteres. Um arquivo de dados de 100.000 caracteres contém somente os caracteres a-f, com as frequências indicadas. Se atribuirmos a cada caractere uma palavra de código de três bits, poderemos codificar o arquivo em 300.000 bits. Usando o código de comprimento variável mostrado, podemos codificar o arquivo em apenas 224.000 bits.

Figura 16.4 Árvores correspondentes aos esquemas de codificação na Figura 16.3. Cada folha é identificada com um caractere e sua frequência de ocorrência. Cada nó interno é identificado com a soma das frequências das folhas em sua subárvore. (**a**) A árvore correspondente ao código de comprimento fixo a = 000, ..., f = 101. (**b**) A árvore correspondente ao código de prefixo ótimo a = 0, b = 101, ..., f = 1100.

Dada uma árvore T correspondente a um código de prefixo, é fácil calcular o número de bits exigidos para codificar um arquivo. Para cada caractere c no alfabeto C, o atributo c.*freq* denota a frequência de c no arquivo e $d_T(c)$ denota a profundidade de folha de c na árvore. Observe que $d_T(c)$ é também o comprimento da palavra de código para o caractere c. Assim, o número de bits exigidos para codificar um arquivo é

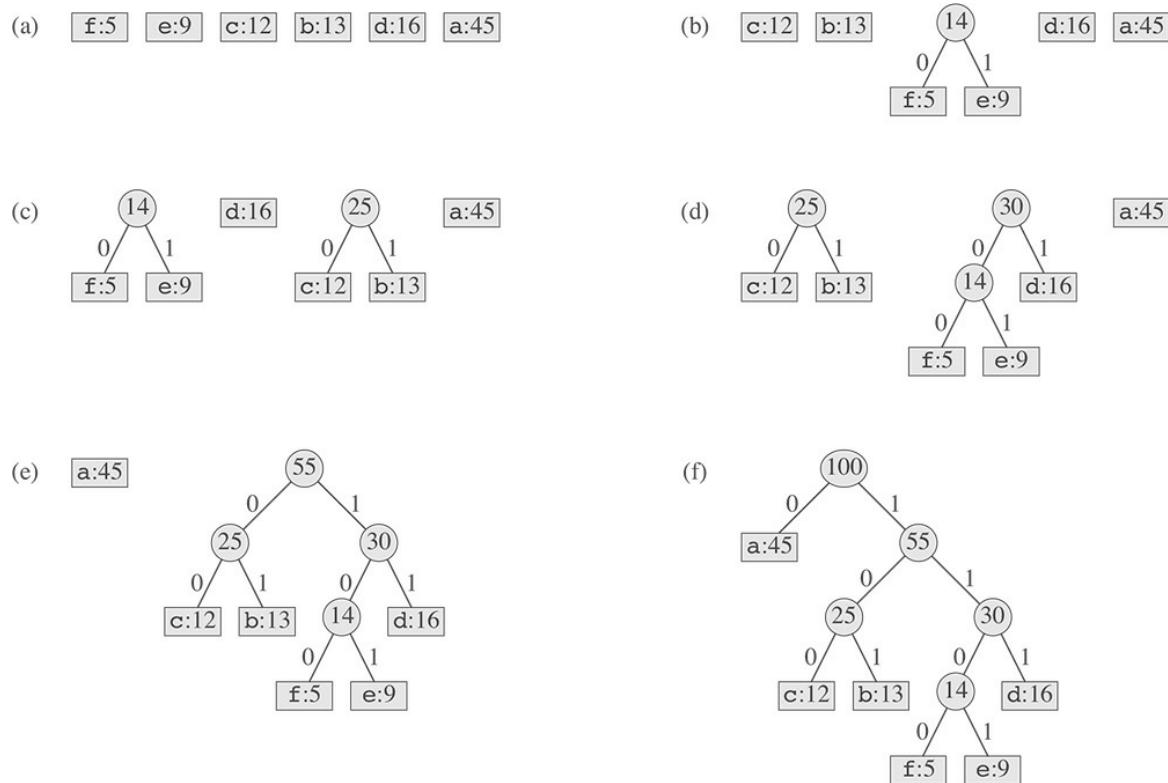
$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)\,, \tag{16.4}$$

que definimos como o ***custo*** da árvore T.

Construção de um código de Huffman

Huffman criou um algoritmo guloso que produz um código de prefixo ótimo denominado ***código de Huffman***. De acordo com nossas observações na Seção 16.2, a prova de sua correção se baseia na propriedade de escolha gulosa e subestrutura ótima. Em vez de demonstrar que essas propriedades são válidas e depois desenvolver pseudocódigo, apresentamos primeiro o pseudocódigo. Isso ajudará a esclarecer como o algoritmo faz escolhas gulosas.

No pseudocódigo a seguir, supomos que C seja um conjunto de n caracteres e que cada caractere $c \in C$ seja um objeto com um atributo $c.freq$ que dá sua frequência. O algoritmo constrói de baixo para cima a árvore T correspondente ao código ótimo. Começa com um conjunto de $|C|$ folhas e executa uma sequência de $|C| - 1$ operações de "intercalação" para criar a árvore final.

O algoritmo usa uma fila de prioridade mínima Q, chaveada no atributo *freq*, para identificar os dois objetos menos frequentes que serão intercalados. Quando intercalamos dois objetos, o resultado é um novo objeto cuja frequência é a soma das frequências dos dois objetos que foram intercalados.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n – 1
4      alocar um novo nó z
5      z.esquerda = x = EXTRACT-MIN(Q)
6      z.direita = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)          // retorna a raiz da árvore.
```

Em nosso exemplo, o algoritmo de Huffman procede como mostra a Figura 16.5. Visto que o alfabeto contém seis letras, o tamanho da fila inicial é $n = 6$, e cinco etapas de intercalação constroem a árvore. A árvore final representa o código de prefixo ótimo. A palavra de código para uma letra é a sequência de etiquetas de arestas no caminho simples da raiz até a letra.

A linha 2 inicializa a fila de prioridade mínima Q com os caracteres em C. O laço **for** nas linhas 3–8 extrai repetidamente da fila os dois nós x e y de frequência mais baixa e os substitui na fila por um novo nó z que representa sua intercalação. A frequência de z é calculada como a soma das frequências de x e y na linha 7. O nó z tem x como seu filho à esquerda e y como seu filho à direita. (Essa ordem é arbitrária; trocar o filho à esquerda pelo filho à direita de qualquer nó produz um código diferente que tem o mesmo custo.) Depois de $n – 1$ intercalações, a linha 9 retorna o único nó que sobrou na fila, que é a raiz da árvore de código.

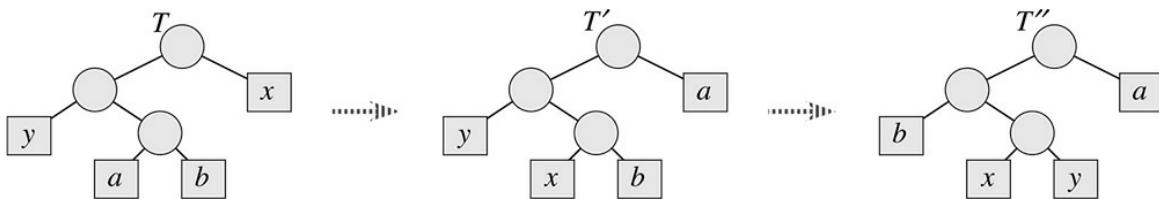
Figura 16.5 Etapas do algoritmo de Huffman para as frequências dadas na Figura 16.3. Cada parte mostra o conteúdo da fila ordenado em ordem crescente de frequência. Em cada etapa, as duas árvores com frequências mais baixas são intercaladas. As folhas são mostradas como retângulos contendo um caractere e sua frequência. Nós internos são mostrados como círculos, contendo a soma das frequências de seus filhos. Uma aresta conectando um nó interno a seus filhos é rotulada por 0 se é uma aresta para um filho à esquerda, e com 1, se é uma aresta para um filho à direita. A palavra de código para uma letra é a sequência de rótulos nas arestas que conectam a raiz à folha correspondente a essa letra. (**a**) O conjunto inicial de $n = 6$ nós, um para cada letra. (**b**)–(**e**) Estágios intermediários. (**f**) A árvore final.

Apesar de que o algoritmo produziria o mesmo resultado se tivéssemos excluído as variáveis x e y — atribuindo diretamente *z.esquerda e z.direita* nas linhas 5 e 6, e mudando a linha 7 para *z.freq = z.esquerda.freq + z.direita.freq* — usaremos os nomes de nó x e y na prova de correção do algoritmo. Portanto, achamos conveniente deixá-los onde estão.

Para analisar o tempo de execução do algoritmo de Huffman, supomos que Q seja implementada como um heap de mínimo binário (veja o Capítulo 6). Para um conjunto C de n caracteres, podemos inicializar Q na linha 2 em tempo $O(n)$ usando o procedimento BUILD-MIN-HEAP discutido na Seção 6.3. O laço **for** nas linhas 3−8 é executado exatamente $n - 1$ vezes e, visto que cada operação de heap requer o tempo $O(\lg n)$, o laço contribui com $O(n \lg n)$ para o tempo de execução. Assim, o tempo de execução total de HUFFMAN em um conjunto de n caracteres é $O(n \lg n)$. Podemos reduzir o tempo de execução para $O(n \lg \lg n)$ substituindo o heap de mínimo binário por uma árvore de van Emde Boas (veja o Capítulo 20).

Correção do algoritmo de Huffman

Para provar que o algoritmo guloso HUFFMAN é correto, mostramos que o problema de determinar um código de prefixo ótimo exibe as propriedades de escolha gulosa e subestrutura ótima. O lema a seguir mostra que a propriedade de escolha gulosa é válida.

Lema 16.2

Seja C um alfabeto no qual cada caractere $c \in C$ tem frequência c.*freq*. Sejam x e y dois caracteres em C que têm as frequências mais baixas. Então, existe um código de prefixo ótimo para C no qual as palavras de código para x e y têm o mesmo comprimento e diferem apenas no último bit.

Prova A ideia da prova é tomar a árvore T que representa um código de prefixo ótimo arbitrário e modificá-la para criar uma árvore que represente outro código de prefixo ótimo, tal que os caracteres x e y apareçam como folhas irmãs de profundidade máxima na nova árvore. Se pudermos construir tal árvore, suas palavras de código para x e y terão o mesmo comprimento e serão diferentes apenas no último bit.

Sejam a e b dois caracteres que são folhas irmãs de profundidade máxima em T. Sem prejuízo da generalidade, supomos que $a.freq \leq b.freq$ e $x.freq \leq y.freq$.

Visto que $x.freq$ e $y.freq$ são as duas frequências de folha mais baixas, em ordem, e $a.freq$ e $b.freq$ são duas frequências arbitrárias, em ordem, temos $x.freq \leq a.freq$ e $y.freq \leq b.freq$.

No restante da prova, é possível que tivéssemos $x.freq = a.freq$ ou $y.freq = b.freq$. Contudo, se tivéssemos $x.freq = b.freq$, teríamos também $a.freq = b.freq = x.freq = y.freq$ (veja o Exercício 16.3-1), e o lema seria trivialmente verdadeiro. Assim, suporemos que $x.freq \neq b.freq$, o que implica em $x \neq b$.

Figura 16.6 Uma ilustração da etapa fundamental na prova do Lema 16.2. Na árvore ótima T, as folhas a e b são duas irmãs de profundidade máxima. As folhas x e y são os dois caracteres que têm as frequências mais baixas; eles aparecem em posições arbitrárias em T. Considerando que $x \neq b$, permutar as folhas a e x produz a árvore T*'*, e permutar as folhas b e y produz a árvore T*''*. Visto que cada permuta não aumenta o custo, a árvore resultante T'' é também uma árvore ótima.

Como a Figura 16.6 mostra, permutamos as posições de *a* e *x* em *T* para produzir uma árvore *T'*, e então permutamos as posições de *b* e *y* em *T'* para produzir uma árvore *T''* na qual *x* e *y* são folhas irmãs de profundidade máxima. (Observe que, se $x = b$, mas $y \neq$ a, então a árvore *T'* não tem *x* e *y* como folhas irmãs de profundidade máxima. Como supomos que $x \neq b$, essa situação não pode ocorrer.) Pela equação (16.4), a diferença de custo entre *T* e *T'* é

$$\begin{aligned} & B(T) - B(T') \\ & \quad = \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) - \\ & \quad = x.freq \cdot d_T(x) + a.freq.d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\ & \quad = x.freq \cdot d_T(x) + a.freq.d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\ & \quad = (a.freq - x.freq)(dT(a) - d_T(x)) \\ & \quad \geq 0, \end{aligned}$$

porque $a.\,freq - x.\,freq$ e $d_T(a) - d_T(x)$ são não negativas. Mais especificamente, $a.freq - x.\,freq$ é não negativa porque x é uma folha de frequência mínima, e $d_T(a) - d_T(x)$ é não negativa porque *a* é uma folha de profundidade máxima em *T*. De modo semelhante, permutar *y* e *b* não aumenta o custo e, assim, $B(T') - B(T'')$ é não negativa. Por consequência, $B(T'') \leq B(T)$, e, visto que *T* é ótima, temos $B(T) \leq B(T'')$, o que implica $B(T'') = B(T)$. Assim, *T''* é uma árvore ótima na qual *x* e *y* aparecem como folhas irmãs de profundidade máxima, da qual o lema decorre.

O Lema 16.2 implica que o processo de construir uma árvore ótima por intercalações pode, sem prejuízo da generalidade, começar com a escolha gulosa de intercalar os dois caracteres de frequência mais baixa. Por que essa é uma escolha gulosa? Podemos ver o custo de uma intercalação isolada como a soma das frequências dos dois itens que estão sendo intercalados. O Exercício 16.3-4 mostra que o custo total da árvore construída é igual à soma dos custos de suas intercalações. De todas as intercalações possíveis em cada etapa, HUFFMAN escolhe aquela que incorre no menor custo.

O próximo lema mostra que o problema de construir códigos de prefixo ótimos tem a propriedade de subestrutura ótima.

Lema 16.3

Seja *C* um dado alfabeto com frequência c.*freq* definida para cada caractere $c \in C$. Sejam *x* e *y* dois caracteres em *C* com frequência mínima. Seja *C'* o alfabeto *C* do qual os caracteres *x* e *y* foram removidos e um novo caractere *z* foi acrescentado, de modo que $C' = C - \{x, y\} \cup \{z\}$. Defina *freq* para *C'* como foi definido para *C*, exceto que $z.\,freq = x.\,freq + y.\,freq$. Seja *T'* qualquer árvore que represente um código de prefixo ótimo para o alfabeto *C'*. Então a árvore *T*, obtida de *T'* pela substituição do nó folha com *z* por um nó interno que tem *x* e *y* como filhos, representa um código de prefixo ótimo para o alfabeto *C*.

Prova Primeiro, mostramos como expressar o custo $B(T)$ da árvore *T* em termos do custo $B(T')$ da árvore *T'*, considerando os custos componentes na equação (16.4). Para cada caractere $c \in C - \{x, y\}$, temos que $d_T(c) = d_{T'}(c)$ e, portanto, $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Visto que $d_T(x) = d_T(y) = d_{T'}(z)$, temos

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

do qual concluímos que

$$\mathrm{B}(T) = \mathrm{B}(T') + x.\,freq + y.\,freq$$

ou, o que é equivalente,

$$\mathrm{B}(T) = \mathrm{B}(\mathrm{T}') - x.\,freq - y.\,freq.$$

Agora, provamos o lema por contradição. Suponha que T não represente um código de prefixo ótimo para C. Então, existe uma árvore T'' tal que $B(T''') < B(T)$. Sem prejuízo da generalidade (pelo Lema 16.2), T'' tem x e y como irmãos. Seja T''' a árvore T'' na qual o pai comum de x e y seja substituído por uma folha z com frequência $z.freq = x.freq + y.freq$. Então,

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T'), \end{aligned}$$

o que produz uma contradição para a suposição de que T' representa um código de prefixo ótimo para C'. Assim, T deve representar um código de prefixo ótimo para o alfabeto C.

Teorema 16.4

O procedimento HUFFMAN produz um código de prefixo ótimo.

Prova Imediata, pelos Lemas 16.2 e 16.3.

Exercícios

16.3-1 Explique por que, na prova do Lema 16.2, se $x.freq = b.freq$, devemos ter $a.freq = b.freq = x.freq = y.freq$.

16.3-2 Prove que uma árvore binária que não é completa não pode corresponder a um código de prefixo ótimo.

16.3-3 Qual é o código de Huffman ótimo para o conjunto de frequências a seguir, baseado nos oito primeiros números de Fibonacci?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Você pode generalizar sua resposta para determinar o código ótimo quando as frequências são os primeiros n números de Fibonacci?

16.3-4 Prove que também podemos expressar o custo total de uma árvore para um código como a soma, para todos os nós internos, das frequências combinadas dos dois filhos do nó.

16.3-5 Prove que, se ordenarmos os caracteres em um alfabeto de modo que suas frequências sejam monotonicamente decrescentes, existe um código ótimo no qual os comprimentos de palavras de código são monotonicamente crescentes.

16.3-6 Suponha que tenhamos um código de prefixo ótimo para um conjunto de caracteres $C = \{0, 1, ..., n - 1\}$ e desejamos transmitir esse código usando o menor número de bits possível. Mostre como representar qualquer código de prefixo ótimo para C usando somente $2n - 1 + n \lg n$ bits. (*Sugestão*: Use $2n - 1$ bits para especificar a estrutura da árvore, como constatada por um percurso da árvore.)

16.3-7 Generalize o algoritmo de Huffman para palavras de código ternárias (isto é, palavras de código que utilizam os símbolos 0, 1 e 2) e prove que ele produz códigos ternários ótimos.

16.3-8 Suponha que um arquivo de dados contenha uma sequência de caracteres de 8 bits tal que todos os 256 caracteres são, quase todos, igualmente comuns: a frequência máxima de caracteres é menor que duas vezes a

frequência mínima de caracteres. Prove que a codificação de Huffman nesse caso não é mais eficiente que usar um código normal de comprimento fixo de 8 bits.

16.3-9 Mostre que nenhum esquema de compressão pode esperar comprimir em um arquivo de caracteres de 8 bits escolhidos aleatoriamente nem um único bit sequer. (*Sugestão*: Compare o número de arquivos possíveis com o número de arquivos codificados possíveis.)

16.4 ★ MATROIDES E MÉTODOS GULOSOS

Nesta seção, esboçamos uma bela teoria para algoritmos gulosos, que descreve muitas situações nas quais o método guloso produz soluções ótimas. Tal teoria envolve estruturas combinatórias conhecidas como "matroides". Embora não abranja todos os casos aos quais um método guloso se aplica (por exemplo, não abrange o problema de seleção de atividades da Seção 16.1 nem o problema de codificação de Huffman da Seção 16.3), ela inclui muitos casos de interesse prático. Além disso, essa teoria foi estendida para abranger muitas aplicações; as notas no final deste capítulo citam referências.

Matroides

Um ***matroide*** é um par ordenado $M = (S, \mathrm{I})$ que satisfaz as seguintes condições: .

1. S é um conjunto finito.
2. é uma família não vazia de subconjuntos de S, denominados subconjuntos ***independentes*** de S, tais que, se $B \in \mathrm{I}$ e $A \subseteq B$, então $A \in \mathrm{I}$. Dizemos que I é ***hereditário*** se satisfaz essa propriedade. Observe que o conjunto vazio $\varnothing$ é necessariamente um membro de I.
3. Se $A \in \mathrm{I}$, $B \in \mathrm{I}$ e $|A| < |B|$, então existe algum elemento $x \in B - A$ tal que $A \cup \{x\} \in \mathrm{I}$. Dizemos que M satisfaz a ***propriedade de troca***.

A palavra "matroide" se deve a Hassler Whitney. Ele estava estudando ***matroides matriciais***, nos quais os elementos de S são as linhas de uma dada matriz e um conjunto de linhas é independente se elas são linearmente independentes no sentido usual. Como o Exercício 16.4-2 pede para mostrar, essa estrutura define um matroide.

Como outro exemplo de matroides, considere o ***matroide gráfico*** $M_G = (S_G, \mathrm{I}_G)$ definido em termos de um determinado grafo não dirigido $G = (V, E)$ da seguinte maneira:

- O conjunto S_G é definido como E, o conjunto de arestas de G.
- Se A é um subconjunto de E, então $A \in \mathrm{I}_G$. se e somente se A é acíclico. Ou seja, um conjunto de arestas A é independente se e somente se o subgrafo $G_A = (V, A)$ forma uma floresta.

O matroide gráfico M_G está intimamente relacionado com o problema da árvore geradora mínima, apresentado em detalhes no Capítulo 23.

Teorema 16.5

Se G = (V, E) é um grafo não dirigido, então $M_G = (S_G, \mathrm{I}_G)$ é um matroide.

Prova Claramente, $S_G = E$ é um conjunto finito. Além disso, é hereditário, já que um subconjunto de uma floresta é uma floresta. Em outros termos, remover arestas de um conjunto acíclico de aresta não pode criar ciclos.

Assim, resta mostrar que M_G satisfaz a propriedade de troca. Suponha que $G_A = (V, A)$ e $G_B = (V, B)$ sejam florestas de G e que $|B| > |A|$. Isto é, A e B são conjuntos acíclicos de arestas e B contém mais arestas que A.

Afirmamos que uma floresta $F = (V_F, E_F)$ contém exatamente $|V_F| - |E_F|$ árvores. Para ver por que, suponha que F consista de três árvores, onde a i-ésima árvore contenha v_i vértices e e_i arestas. Então, temos

$$
\begin{aligned}
|E_F| &= \sum_{i=1}^{t} e_i \\
&= \sum_{i=1}^{t} (v_i - 1) \quad \text{(pelo Teorema B.2)} \\
&= \sum_{i=1}^{t} v_i - t \\
&= |V_F| - t\,,
\end{aligned}
$$

que implica que $t = |V_F| - |E_F|$. Assim, a floresta G_A contém $|V| - |A|$ árvores, e a floresta G_B contém $|V| - |B|$ árvores.

Visto que a floresta G_B tem menos árvores que a floresta G_A, a floresta G_B deve conter alguma árvore T cujos vértices estão em duas árvores diferentes na floresta G_A. Além disso, como T está conectada, ela deve conter uma aresta (u, v) tal que os vértices u e v estão em árvores diferentes na floresta G_A. Visto que a aresta (u, v) conecta vértices em duas árvores diferentes na floresta G_A, podemos adicionar a aresta (u, v) à floresta G_A sem criar um ciclo. Então, M_G satisfaz a propriedade de troca, completando a prova de que M_G é um matroide.

Dado um matroide $M = (S, I)$, dizemos que um elemento $x \notin A$ é uma ***extensão*** de $A \in I$ se x pode ser acrescentado a A preservando independência; isto é, x é uma extensão de A se $A \cup \{x\} \in I$. Como exemplo, considere um matroide gráfico M_G. Se A é um conjunto independente de arestas, então a aresta e é uma extensão de A se e somente se e não está em A e o acréscimo de x a A não criar um ciclo.

Se A é um subconjunto independente em um matroide M, dizemos que A é ***maximal*** se não tem nenhuma extensão. Isto é, A é maximal se não está contido em nenhum subconjunto independente maior de M. A propriedade dada a seguir é muito útil.

Teorema 16.6

Todos os subconjuntos independentes maximais em um matroide têm o mesmo tamanho.

Prova Suponha, por contradição, que A seja um subconjunto independente maximal de M e que exista um outro subconjunto independente maximal maior B de M. Então, a propriedade de troca implica que, para algum $x \in B - A$, podemos estender A até um conjunto independente maior $A \cup \{x\}$, contradizendo a hipótese de que A é maximal.

Como ilustração desse teorema, considere um matroide gráfico M_G para um grafo conexo e não dirigido G. Todo subconjunto independente máximo de M_G deve ser uma árvore livre com exatamente $|V| - 1$ arestas que conecta todos os vértices de G. Tal árvore é denominada ***árvore geradora*** de G.

Dizemos que um matroide $M = (S, I)$ é ***ponderado*** se está associado a uma função peso w que atribui um peso estritamente positivo $w(x)$ a cada elemento $x \in S$. A função peso w se estende a subconjuntos de S por somatório:

$$w(A) = \sum_{x \in A} w(x)$$

para qualquer $A \subseteq S$. Por exemplo, se $w(e)$ denota o peso de uma aresta e em um matroide gráfico M_G, então $w(A)$ é o peso total das arestas no conjunto de arestas A.

Algoritmos gulosos em um matroide ponderado

Muitos problemas para os quais uma abordagem gulosa dá soluções ótimas podem ser formulados em termos de encontrar um subconjunto independente de peso máximo em um matroide ponderado. Isto é, temos um matroide ponderado $M = (S, I)$ e desejamos encontrar um conjunto independente $A \in I$ tal que $w(A)$ seja maximizado. Denominamos tal subconjunto, que é independente e tem peso máximo possível, subconjunto ***ótimo*** do matroide. Como o peso $w(x)$ de qualquer elemento $x \in S$ é positivo, um subconjunto ótimo é sempre um subconjunto independente maximal — ele sempre ajuda a tornar A tão grande quanto possível.

Por exemplo, no ***problema da árvore geradora mínima***, temos um grafo conexo não dirigido $G = (V, E)$ e uma função comprimento w tal que $w(e)$ é o comprimento (positivo) da aresta e. (Usamos o termo "comprimento" aqui em referência aos pesos de arestas originais para o grafo, reservando o termo "peso" para nos referirmos aos pesos no matroide associado.) Devemos encontrar um subconjunto das arestas que conecte todos os vértices e tenha comprimento total mínimo. Para ver esse problema como um problema de encontrar um subconjunto ótimo de um matroide, considere o matroide ponderado M_G com função peso w', onde $w'(e) = w_0 - w(e)$ e w_0 é maior que o comprimento máximo de qualquer aresta. Nesse matroide ponderado, todos os pesos são positivos e um subconjunto ótimo é uma árvore geradora de comprimento total mínimo no grafo original. Mais especificamente, cada subconjunto independente máximo A corresponde a uma árvore geradora com $|V| - 1$ arestas e, visto que

$$\begin{aligned} w'(A) &= \sum_{e \in A} w'(e) \\ &= \sum_{e \in A} (w_0 - w(e)) \\ &= (|V|-1)w_0 - \sum_{e \in A} w(e) \\ &= (|V|-1)w_0 - w(A) \end{aligned}$$

para qualquer subconjunto independente máximo A, um subconjunto independente que maximiza a quantidade $w'(A)$ deve minimizar $w(A)$. Assim, qualquer algoritmo que pode encontrar um subconjunto ótimo A em um matroide arbitrário pode resolver o problema da árvore geradora mínima.

O Capítulo 23 dá algoritmos para o problema da árvore geradora mínima, mas aqui daremos um algoritmo guloso que funciona para qualquer matroide ponderado. O algoritmo toma como entrada um matroide ponderado $M = (S, I)$ com uma função peso positivo associada w e retorna um subconjunto ótimo A. Em nosso pseudocódigo, denotamos os componentes de M por $M.S$ e $M.I$, e a função peso por w. O algoritmo é guloso porque considera cada elemento $x \in S$ por vez em ordem monotonicamente decrescente de peso e o adiciona imediatamente ao conjunto A que está sendo acumulado, se $A \cup \{x\}$ é independente.

```
GREEDY(M, w)
1  A = ∅
2  ordenar M.S em sequência monotonicamente decrescente de peso w
3  for cada x ∈ M.S, tomado em ordem monotonicamente decrescente de peso w(x)
4        if A ∪ {x} ∈ M.I
5           A = A ∪ {x}
6  return A
```

A linha 4 verifica se acrescentar cada elemento x a A manteria A como um conjunto independente. Se A permanecer independente, a linha 5 acrescentará x a A. Caso contrário, x é descartado. Visto que o conjunto vazio é

independente, e já que cada iteração do laço **for** mantém a independência de A, o subconjunto A é sempre independente, por indução. Consequentemente, GREEDY sempre retorna um subconjunto independente A. Veremos em breve que A é um subconjunto cujo peso é o máximo possível e, portanto, A é um subconjunto ótimo.

O tempo de execução de GREEDY é fácil de analisar. Considere que n denota $|S|$. A fase de ordenação de GREEDY demora o tempo $O(n \lg n)$. A linha 4 é executada exatamente n vezes, uma vez para cada elemento de S. Cada execução da linha 4 requer verificar se o conjunto $A \cup \{x\}$ é ou não independente. Se cada verificação demorar o tempo $O(f(n))$, o algoritmo inteiro será executado no tempo $O(n \lg n + nf(n))$.

Agora, provaremos que GREEDY retorna um subconjunto ótimo.

Lema 16.7 (Matroides exibem a propriedade de escolha gulosa)

Suponha que $M = (S, I)$ seja um matroide ponderado com função peso w e que S esteja ordenado em ordem monotonicamente decrescente de peso. Seja x o primeiro elemento de S tal que $\{x\}$ seja independente se tal x existir. Se x existe, então existe um subconjunto ótimo A de S que contém x.

Prova Se nenhum tal x existe, então o único subconjunto independente é o conjunto vazio e o lema é verdadeiro por vacuidade. Senão, seja B qualquer subconjunto ótimo não vazio. Suponha que $x \notin B$; caso contrário, fazer $A = B$ dá um subconjunto ótimo de S que contém x.

Nenhum elemento de B tem peso maior que $w(x)$. Para ver por que, observe que $y \in B$ implica que $\{y\}$ é independente, já que $B \in I$, e I é hereditário. Portanto, nossa escolha de x assegura que $w(x) \geq w(y)$ para qualquer $y \in B$.

Construa o conjunto A como descrevemos a seguir. Comece com $A = \{x\}$. Pela escolha de x, o conjunto A é independente. Usando a propriedade de troca, determine repetidamente um novo elemento de B que podemos acrescentar a A até $|A| = |B|$ e ao mesmo tempo preservar a independência de A. Nesse ponto, A e B são iguais exceto que A tem x e B tem algum outro elemento y. Isto é, $A = B - \{y\} \cup \{x\}$ para algum $y \in B$, e assim

$$\begin{aligned} w(A) &= w(B) - w(y) + w(x) \\ &\geq w(B). \end{aligned}$$

Como o conjunto B é ótimo, o conjunto A, que contém x, também deve ser ótimo.

Mostraremos em seguida que, se um elemento não é uma opção inicialmente, não poderá ser uma opção mais tarde.

Lema 16.8

Seja $M = (S, I)$ um matroide. Se x é um elemento de S que é uma extensão de algum subconjunto independente A de S, então x também é uma extensão de /0.

Prova Como x é uma extensão de A, temos que $A \cup \{x\}$ é independente. Como é hereditário, $\{x\}$ tem de ser independente. Assim, x é uma extensão de /0.

Corolário 16.9

Seja $M = (S, I)$ um matroide. Se x é um elemento de S tal que x não é uma extensão de /0, então x não é uma extensão de nenhum subconjunto independente A de S.

Prova Este corolário é simplesmente o contrapositivo do Lema 16.8.

O Corolário 16.9 diz que qualquer elemento que não pode ser usado imediatamente nunca poderá ser usado. Então, GREEDY não pode gerar um erro por ignorar quaisquer elementos iniciais em S que não sejam uma extensão de /0, já que eles nunca poderão ser usados.

Lema 16.10 (Matroides exibem a propriedade de subestrutura ótima)

Seja x o primeiro elemento de S escolhido por GREEDY para o matroide ponderado $M = (S, I)$. O problema restante de determinar um subconjunto independente de peso máximo contendo x se reduz a determinar um subconjunto independente de peso máximo do matroide ponderado $M' = (S', I')$, onde

$$\begin{aligned} S' &= \{y \in S : \{x, y\} \in \mathcal{I}\}, \\ I' &= \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\}, \end{aligned}$$

e a função peso para M' é a função peso para M, restrita a S'. (Denominamos M' a ***contração*** de M pelo elemento x.)

Prova Se A é um subconjunto independente de peso máximo de M contendo x, então $A' = A - \{x\}$ é um subconjunto independente de M'. Reciprocamente, qualquer subconjunto independente A' de M' produz um subconjunto independente $A = A' \cup \{x\}$ de M. Visto que em ambos os casos temos que $w(A) = w(A') + w(x)$, uma solução de peso máximo em M contendo x produz uma solução de peso máximo em M' e vice-versa.

Teorema 16.11 (Correção do algoritmo guloso em matroides)

Se $M = (S, I)$ é um matroide ponderado com função peso w, então GREEDY(M, w) retorna um subconjunto ótimo.

Prova Pelo Corolário 16.9, quaisquer elementos que GREEDY ignorar inicialmente porque não são extensões de /0 podem ser esquecidos, já que nunca serão úteis. Tão logo GREEDY selecione o primeiro elemento x, o Lema 16.7 implica que o algoritmo não erra por acrescentar x a A, já que existe um subconjunto ótimo contendo x. Por fim, o Lema 16.10 implica que o problema restante é o de encontrar um subconjunto ótimo no matroide M' que seja a contração de M por x. Depois que o procedimento GREEDY atribui A com $\{x\}$,podemos interpretar que todas as suas etapas restantes agem no matroide $M' = (S', 0/')$ porque B é independente em M' se e somente se $B \cup \{x\}$ é independente em M, para todos os conjuntos $B \in I'$. Assim, a operação subsequente de GREEDY encontrará um subconjunto independente de peso máximo para M', e a operação global de GREEDY encontrará um subconjunto independente de peso máximo para M.

Exercícios

16.4-1 Mostre que (S, I_k) é um matroide, onde S é qualquer conjunto finito e I_k é o conjunto de todos os subconjuntos de S de tamanho no máximo k, onde $k \leq |S|$.

16.4-2 ★ Dada uma matriz T $m \cdot n$ em algum corpo (como os números reais), mostre que (S, I) é um matroide, onde S é o conjunto de colunas de T e $A \in I$ se e somente se as colunas em A são linearmente independentes.

16.4-3 ★ Mostre que, se (S, I) é um matroide, então (S, I') é um matroide, onde

$I' = \{A' : S - A'$ contém algum $A \in I$ máximo$\}$.

Isto é, os conjuntos independentes maximais de (S, I') são exatamente os complementos dos conjuntos independentes maximais de (S, I').

16.4-4 ★ Seja S um conjunto finito e seja $S_1, S_2 \dots S_k$ uma partição de S em subconjuntos disjuntos não vazios. Defina a estrutura (S, I) pela condição de que $I = \{A : |A \cap S| \leq 1$ para $i = 1, 2, \dots, k\}$. Mostre que (S, I) é um

matroide. Isto é, o conjunto de todos os conjuntos

A que contêm no máximo um membro em cada subconjunto na partição determina os conjuntos independentes de um matroide.

16.4-5 Mostre como transformar a função peso de um problema de matroide ponderado, onde a solução ótima desejada é um subconjunto independente maximal de *peso mínimo*, para fazer dele um problema-padrão de matroide ponderado. Mostre cuidadosamente que sua transformação está correta.

16.5 ★ UM PROBLEMA DE PROGRAMAÇÃO DE TAREFAS COMO UM MATROIDE

Um problema interessante que podemos resolver utilizando matroides é o de programar otimamente tarefas de tempo unitário em um único processador, onde cada tarefa tem um prazo final e uma multa que deve ser paga se esse prazo final não for cumprido. O problema parece complicado, mas podemos resolvê-lo de um modo surpreendentemente simples expressando-o como um matroide e utilizando um algoritmo guloso.

Uma ***tarefa de tempo unitário*** é um trabalho, como um programa a ser executado em um computador, que requer exatamente uma unidade de tempo para ser concluído. Dado um conjunto finito S de tarefas de tempo unitário, uma ***programação*** para S é uma permutação de S especificando a ordem em que essas tarefas devem ser executadas. A primeira tarefa na programação começa no tempo 0 e termina no tempo 1; a segunda tarefa começa no tempo 1 e termina no tempo 2, e assim por diante.

O problema de ***programar tarefas de tempo unitário com prazos finais e multas para um único processador*** tem as seguintes entradas:

- um conjunto $S = \{a_1, a_2, ..., a_n\}$ de n tarefas de tempo unitário;
- um conjunto de n ***prazos finais*** inteiros $d_1, d_2, ..., d_n$, tal que cada d_i satisfaz $1 \leq d_i \leq n$ e a tarefa a_1 deve terminar até o tempo d_i;
- um conjunto de n pesos não negativos ou ***multas*** $w_1, w_2, ..., w_n$, tal que incorremos em uma multa w_i se a tarefa a_i não termina até o tempo d_i e não incorremos em nenhuma multa se uma tarefa termina em seu prazo final.

Queremos determinar uma programação para S que minimize a multa total incorrida quando prazos finais não são cumpridos.

Considere uma dada programação. Dizemos que uma tarefa está ***atrasada*** nessa programação se ela termina após seu prazo final. Caso contrário, a tarefa está ***adiantada*** na programação. Sempre podemos transformar uma programação arbitrária para a ***forma adiantada*** na qual as tarefas adiantadas precedem as tarefas atrasadas. Para ver isso, observe que, se alguma tarefa adiantada a_i seguir alguma tarefa atrasada a_j, poderemos trocar as posições de a_i e a_j, e a_i ainda estará adiantada e a_j ainda estará atrasada.

Além do mais, podemos afirmar que sempre podemos transformar uma programação arbitrária para a ***forma canônica***, na qual as tarefas adiantadas precedem as tarefas atrasadas e programamos as tarefas adiantadas em ordem monotonicamente crescentes de prazos finais. Para tal, colocamos a programação na forma adiantada. Então, desde que existam duas tarefas adiantadas a_i *e* a_j terminando nos tempos respectivos k e $k + 1$ na programação, tais que $d_j < d_i$, trocamos as posições de a_i *e* a_j. Como a tarefa a_j está adiantada antes da troca, $k + 1 \leq d_j$. Portanto, $k + 1 < d_i$ e, assim, a tarefa a_i ainda está adiantada após a troca. Como a tarefa a_j é movida antes na programação, permanece adiantada após a troca.

Assim, a busca por uma programação ótima se reduz a encontrar um conjunto A de tarefas que designamos como adiantadas na programação ótima. Tendo determinado A, podemos criar a programação propriamente dita organizando uma lista dos elementos de A em ordem monotonicamente crescente de prazo final e, em seguida, organizando uma lista de tarefas atrasadas (isto é, $S - A$) em qualquer ordem, produzindo uma ordenação canônica da programação ótima.

Dizemos que um conjunto A de tarefas é ***independente*** se existe uma programação para essas tarefas tal que nenhuma tarefa está atrasada. É claro que o conjunto de tarefas adiantadas para uma programação forma um conjunto

independente de tarefas. Seja I o conjunto de todos os conjuntos independentes de tarefas.

Considere o problema de determinar se um dado conjunto A de tarefas é independente. Para $t = 0, 1, 2, ..., n$, seja $N_t(A)$ o número de tarefas em A cujo prazo final é t ou mais cedo. Observe que $N_0(A) = 0$ para qualquer conjunto A.

Lema 16.12

Para qualquer conjunto de tarefas A, as declarações a seguir são equivalentes.

1. O conjunto A é independente.
2. Para $t = 0, 1, 2, ..., n$, temos $N_t(A) \leq t$.
3. Se as tarefas em A estão programadas em ordem monotonicamente crescente de prazos finais, então nenhuma tarefa está atrasada.

Prova Para mostrar que (1) implica (2), provamos o contrapositivo: se $N_t(A) > t$ para algum t, então não existe nenhum modo de fazer uma programação sem nenhuma tarefa atrasada para o conjunto A porque mais de t tarefas devem terminar antes do tempo t. Por essa razão, (1) implica (2). Se (2) é válida, então (3) deve ser válida: não existe nenhum modo de "ficarmos emperrados" quando programamos tarefas em ordem monotonicamente crescente de prazos finais, já que (2) implica que o i-ésimo maior prazo final é no mínimo i. Finalmente, (3) implica trivialmente (1).

Usando a propriedade 2 do Lema 16.12, podemos calcular facilmente se um dado conjunto de tarefas é independente ou não (veja o Exercício 16.5-2).

O problema de minimizar a soma das multas por tarefas atrasadas é igual ao problema de maximizar a soma das multas das tarefas adiantadas. Assim, o teorema a seguir assegura que podemos usar o algoritmo guloso para encontrar um conjunto independente A de tarefas com a multa total máxima.

Teorema 16.13

Se S é um conjunto de tarefas de tempo unitário com prazos finais e se I é o conjunto de todos os conjuntos de tarefas independentes, então o sistema correspondente (S, I) é um matroide.

Prova Todo subconjunto de um conjunto independente de tarefas certamente é independente. Para provar a propriedade de troca, suponha que B e A sejam conjuntos de tarefas independentes e que $|B| > |A|$. Seja k o maior t tal que $N_t(B) \leq N_t(A)$. (Tal valor de t existe, já que $N_0(A) = N_0(B) = 0$.) Como $N_n(B) = |B|$ e $N_n(A) = |A|$, mas $|B| > |A|$, devemos ter que $k < n$ e que $N_j(B) > N_j(A)$ para todo j na faixa $k + 1 \leq j \leq n$. Portanto, B contém mais tarefas com prazo final $k + 1$ do que A. Seja ai uma tarefa em $B - A$ com prazo final $k + 1$. Seja $A' = A \cup \{a_i\}$.

Agora, mostraremos que A' deve ser independente, usando a propriedade 2 do Lema 16.12. Para $0 \leq t \leq k$, temos $N_t(A') = N_t(A) \leq t$, já que A é independente. Para $k < t \leq n$, temos $N_t(A') \leq N_t(B) \leq t$, já que B é independente. Portanto, A' é independente, o que conclui nossa prova de que (S, I) é um matroide.

Pelo Teorema 16.11, podemos usar um algoritmo guloso para encontrar um conjunto independente de peso máximo de tarefas A. Então, podemos criar uma programação ótima na qual as tarefas em A sejam suas tarefas adiantadas. Esse método é um algoritmo eficiente para programação de tarefas de tempo unitário com prazos finais e multas para um único processador. O tempo de execução é $O(n_2)$ utilizando GREEDY, já que cada uma das $O(n)$ verificações de independência feitas por esse algoritmo demora o tempo $O(n)$ (veja o Exercício 16.5-2). O Problema 16-4 dá uma implementação mais rápida.

Tarefa							
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figura 16.7 Uma instância do problema de programar tarefas de tempo unitário com prazos finais e multas para um único processador.

A Figura 16.7 demonstra um exemplo do problema de programação de tarefas de tempo unitário com prazos finais e multas para um único processador. Nesse exemplo, o algoritmo guloso seleciona as tarefas a_1, a_2, a_3 e a_4 em ordem, depois rejeita a_5 (porque $N_4(\{a_1, a_2, a_3, a_4, a_5\}) = 5$) e a_6 (porque $N4(\{a_1, a_2, a_3, a_4, a_6\}) = 5$), e finalmente aceita a tarefa a_7. A programação final ótima é

$$\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle,$$

que incorre em uma multa total de $w_5 + w_6 = 50$.

Exercícios

16.5-1 Resolva a instância do problema de programação dado na Figura 16.7, mas substituindo cada multa w_i por $80 - w_i$.

16.5-2 Mostre como usar a propriedade 2 do Lema 16.12 para determinar no tempo $O(|A|)$ se um dado conjunto A de tarefas é independente ou não.

Problemas

16-1 ***Troco em moedas***

Considere o problema de dar troco para n centavos usando o menor número de moedas. Suponha que o valor de cada moeda seja um inteiro.

a. Descreva um algoritmo guloso para dar um troco utilizando moedas de 25 centavos, 10 centavos, 5 centavos e 1 centavo. Prove que seu algoritmo produz uma solução ótima.

b. Suponha que as moedas disponíveis estejam nas denominações que são potências de c, isto é, as denominações são $c^0, c^1, \ldots, c^k$ para alguns inteiros $c > 1$ e $k \geq 1$. Mostre que o algoritmo guloso sempre produz uma solução ótima.

c. Dê um conjunto de denominações de moedas para o qual o algoritmo guloso não produz uma solução ótima. Seu conjunto deve incluir um centavo, de modo que exista uma solução para todo valor de n.

d. Dê um algoritmo de tempo $O(nk)$ que dê o troco para qualquer conjunto de k denominações diferentes de moeda, considerando que uma das moedas é 1 centavo.

16-2 Programação para minimizar o tempo médio de conclusão

Suponha que você tenha um conjunto $S = \{a_1, a_2, ..., a_n\}$ de tarefas, no qual a tarefa a_i requeira p_i unidades de tempo de processamento para ser concluída, desde que seja iniciada. Você tem um computador no qual executar essas tarefas, e o computador só pode executar uma tarefa por vez. Seja c_i o ***tempo de conclusão*** da tarefa a_i, isto é, o tempo no qual o processamento da tarefa a_i é concluído. Sua meta é minimizar o tempo médio de conclusão, ou seja, minimizar $(1/n)\sum_{i-1}^{n} c_i$. Por exemplo suponha que haja duas tarefas, a_1 e a_2, com $p_1 = 3$ e $p_2 = 5$, e considere a programação na qual a_2 é executada primeiro, seguida por a_1. Então, $c_2 = 5$, $c_1 = 8$, e o tempo médio de conclusão é $(5 + 8)/2 = 6{,}5$. Todavia, se a tarefa a_1 for executada primeiro, $c_1 = 3$, $c_2 = 8$, e o tempo médio de conclusão é $(3 + 8)/2 = 5{,}5$.

a. Dê um algoritmo que programe as tarefas de modo a minimizar o tempo médio de conclusão. Cada tarefa deve ser executada de modo não preemptivo, isto é, uma vez iniciada a tarefa a_i, ela deve ser executada continuamente durante p_i unidades de tempo. Prove que seu algoritmo minimiza o tempo médio de conclusão e informe o tempo de execução do seu algoritmo.

b. Suponha agora que as tarefas não estejam todas disponíveis ao mesmo tempo. Isto é, nenhuma tarefa pode começar antes de seu ***tempo de liberação*** r_i. Suponha também que permitimos a ***preempção***, de modo que uma tarefa pode ser suspensa e reiniciada mais tarde. Por exemplo, uma tarefa a_i com tempo de processamento $p_i = 6$ e tempo de liberação $r^i = 1$ poderia iniciar sua execução no tempo 1 e ser suspensa no tempo 4. Então, ela poderia recomeçar no tempo 10, mas ser suspensa no tempo 11 e, por fim, recomeçar no tempo 13 e concluir no tempo 15. A tarefa a_i é executada durante um total de seis unidades de tempo, mas seu tempo de execução foi dividido em três partes. Com isso, o tempo de conclusão de a_i é 15. Dê um algoritmo que programe as tarefas de modo a minimizar o tempo médio de conclusão nesse novo cenário. Prove que seu algoritmo minimiza o tempo médio de conclusão e informe o tempo de execução do seu algoritmo.

16-3 >Subgrafos acíclicos

a. A ***matriz de incidência*** para um grafo não dirigido $G = (V, E)$ é uma matriz $|V| \cdot |E|$ M tal que $M_{ve} = 1$ se a aresta e incidir no vértice v; caso contrário, $M_{ve} = 0$. Mostre que um conjunto de colunas de M é linearmente independente no corpo dos inteiros módulo 2 se e somente se o conjunto correspondente de arestas é acíclico. Então, use o resultado do Exercício 16.4-2 para dar uma prova alternativa de que (E, I) da parte (a) é um matroide.

b. Suponha que associamos um peso não negativo $w(e)$ a cada aresta em um grafo não dirigido $G = (V, E)$. Dê um algoritmo eficiente para encontrar um subconjunto acíclico de E de peso total máximo.

c. Seja $G(V, E)$ um grafo dirigido arbitrário e seja (E, I) definido de tal modo que A ∈ se e somente se A não contém nenhum ciclo dirigido. Dê um exemplo de um grafo dirigido G tal que o sistema associado (E, I.) não seja um matroide. Especifique qual condição de definição para um matroide deixa de ser válida.

d. A ***matriz de incidência*** para um grafo dirigido $G = (V, E)$ sem nenhum laço é uma matriz $|V| \times |E|$ M tal que $M_{ve} = -1$ se a aresta e sai do vértice v, $M_{ve} = 1$ se a aresta e entra no vértice v e $M_{ve} = 0$ em qualquer outro caso. Mostre que, se um conjunto de colunas de M é linearmente independente, o conjunto correspondente de arestas não contém um ciclo dirigido.

e. O Exercício 16.4-2 nos diz que o conjunto de conjuntos linearmente independentes de colunas de qualquer matriz M forma um matroide. Explique cuidadosamente por que os resultados das partes (c) e

(d) não são contraditórios. Como poderia deixar de haver uma perfeita correspondência entre a noção de que um conjunto de arestas é acíclico e a noção de que um conjunto associado de colunas da matriz de incidência é linearmente independente?

16-4 Variações de programação

Considere o algoritmo a seguir para resolver o problema da Seção 16.5 de programar tarefas de tempo unitário com prazos finais e multas. Sejam todos os n intervalos de tempo inicialmente vazios, onde um intervalo de tempo i é o intervalo de tempo de comprimento unitário que termina no tempo i. Consideramos as tarefas em ordem monotonicamente decrescente de multa. Quando consideramos a tarefa a_j, se existir um intervalo de tempo no prazo final d_j de a_j ou antes dele que ainda esteja vazio, atribua a_j a tal intervalo mais recente, preenchendo-o. Se tal intervalo de tempo não existir, atribua a tarefa a_j ao intervalo de tempo mais recente ainda não preenchido.

a. Mostre que esse algoritmo sempre dá uma resposta ótima.

b. Use a floresta rápida de conjuntos disjuntos apresentada na Seção 21.3 para implementar o algoritmo eficientemente. Considere que o conjunto de tarefas de entrada já esteja ordenado em ordem monotonicamente decrescente de multa. Analise o tempo de execução de sua implementação.

16-5 Caching Off-line

Computadores modernos usam uma cache para armazenar uma pequena quantidade de dados em memória rápida. Embora um programa possa acessar grande quantidade de dados, armazenar um pequeno subconjunto da memória principal na ***cache*** — uma memória pequena, porém rápida — pode resultar em grande redução do tempo de acesso global. Quando executado, um programa de computador faz uma sequência $\langle r_1, r_2, ..., r_n \rangle$ de n solicitações à memória, e cada uma dessas solicitações é para um determinado elemento de dados. Por exemplo, um programa que acessa 4 elementos distintos $\{a, b, c, d\}$ poderia fazer a sequência de solicitações $\langle d, b, d, b, d, a, c, d, b, a, c, b \rangle$. Seja k o tamanho da cache. Quando ela contém k elementos e o programa solicita o $(k + 1)$-ésimo elemento, o sistema tem de decidir, para essa solicitação e para cada uma das solicitações subsequentes, quais k elementos manter na cache. Mais exatamente, para cada solicitação r_i, o algoritmo de gerenciamento de cache verifica se o elemento r_i já está na cache. Se está, temos uma **presença na cache**; senão, temos uma **ausência da cache**. Quando ocorre uma ausência da cache, o sistema retira r_i da memória principal e o algoritmo de gerenciamento da cache deve decidir se mantém r_i na cache. Se decidir manter r_i e a cache já contiver k elementos, o algoritmo tem de excluir um elemento para dar lugar a r_i. O algoritmo de gerenciamento da cache exclui dados com o objetivo de minimizar o número de ausências da cache para a seção de solicitações inteira.

Normalmente, caching é um problema on-line. Isto é, temos de tomar decisões sobre quais dados manter na cache sem conhecer as solicitações futuras. Entretanto, aqui, consideramos a versão off-line desse problema, na qual temos de antemão toda a sequência de n solicitações e o tamanho da cache k, e desejamos minimizar o número total de ausências da cache.

Podemos resolver esse problema off-line por uma estratégia gulosa denominada ***futuro mais longínquo***, que escolhe excluir o item presente na cache cujo próximo acesso na sequência de solicitações ocorre no futuro mais longínquo.

a. Escreva pseudocódigo para um gerenciador de cache que usa a estratégia do futuro mais longínquo. A entrada deve ser uma sequência $<r_1, r_2, ..., r_n>$ de solicitações e um tamanho de cache k; a saída deve ser uma sequência de decisões sobre qual elemento de dado (se houver algum) eliminar a cada solicitação.

Qual é o tempo de execução do seu algoritmo?

b. Mostre que o problema de caching off-line exibe subestrutura ótima.

c. Prove que a estratégia futuro mais longínquo produz o menor número possível de ausências da cache.

NOTAS DO CAPÍTULO

Uma quantidade muito maior de material sobre algoritmos gulosos e matroides pode ser encontrada em Lawler [224] e em Papadimitriou e Steiglitz [271].

O algoritmo guloso apareceu pela primeira vez na literatura de otimização combinatória em um artigo de 1971 por Edmonds [101], embora a teoria de matroides date de um artigo de 1935 por Whitney [355].

Nossa prova da correção do algoritmo guloso para o problema de seleção de atividades se baseia na de Gavril [131]. O problema de programação de tarefas é estudado em Lawler [224], Horowitz, Sahni e Rajasekaran [181]; e Brassard e Bratley [55].

Os códigos de Huffman foram criados em 1952 [185]; Lelewer e Hirschberg [231] pesquisaram técnicas de compressão de dados conhecidas desde 1987.

Uma extensão da teoria de matroides para a teoria de *guloides* (*greedoids*, em inglês) teve como pioneiros Korte e Lovász [216, 217, 218, 219], que generalizam bastante a teoria apresentada aqui.

---

[1] Às vezes, nos referimos aos conjuntos S como subproblemas em vez de apenas conjuntos de atividades. Sempre ficará claro pelo contexto se estamos nos referindo a S_k como um conjunto de atividades ou como um subproblema cuja entrada é aquele conjunto.

[2] Como o pseudocódigo toma *s* e *f* como arranjos, é indexado a eles com chaves em vez de subscritos.

[3] Talvez "códigos livres de prefixo" fosse um nome melhor, mas a expressão "códigos de prefixo" é padrão na literatura.

17 ANÁLISE AMORTIZADA

Em uma ***análise amortizada***, calculamos a média do tempo requerido para executar uma sequência de operações de estruturas de dados em todas as operações executadas. Com análise amortizada, podemos mostrar que o custo médio de uma operação é pequeno, se calculada a média de uma sequência de operações, ainda que uma única operação dentro da sequência possa ser custosa. A análise amortizada é diferente da análise do caso médio porque não envolve probabilidade; uma análise amortizada garante o *desempenho médio de cada operação no pior caso*.

As três primeiras seções deste capítulo abordam as três técnicas mais comuns usadas em análise amortizada. A Seção 17.1 começa com a análise agregada, na qual determinamos um limite superior $T(n)$ para o custo total de uma sequência de n operações. Depois, o custo amortizado por operação é $T(n)/n$. Adotamos o custo médio como o custo amortizado de cada operação, de modo que todas as operações têm o mesmo custo amortizado.

A Seção 17.2 focaliza o método de contabilidade, no qual determinamos um custo amortizado de cada operação. Quando há mais de um tipo de operação, cada tipo de operação pode ter um custo amortizado diferente. O método de contabilidade cobra a mais por algumas operações no início da sequência e armazena essa quantia a mais como "crédito pré-pago" para objetos específicos na estrutura de dados. Mais adiante na sequência, o crédito paga operações que foram cobradas a menor que seu custo real.

A Seção 17.3 discute o método do potencial, que é semelhante ao método de contabilidade no sentido de que determinamos o custo amortizado de cada operação e podemos cobrar a mais por operações no início para mais tarde compensar cobranças a menor. O método do potencial mantém o crédito como a "energia potencial" da estrutura de dados como um todo, em vez de associar o crédito a objetos individuais dentro da estrutura de dados.

Usaremos dois exemplos para examinar esses três métodos. Um é uma pilha com a operação adicional MULTIPOP, que retira de uma pilha vários objetos de uma vez. O outro é um contador binário que conta a partir de 0 por meio da operação isolada INCREMENT.

Ao ler este capítulo, tenha em mente que as cobranças atribuídas durante uma análise amortizada servem apenas para a análise. Elas não precisam — e não devem — aparecer no código. Se, por exemplo, atribuirmos um crédito a um objeto x quando usamos o método de contabilidade, não precisamos atribuir um valor adequado a algum atributo no código, tal como *x.crédito*. Muitas vezes a análise amortizada nos dá uma certa percepção de uma determinada estrutura de dados e essa percepoção pode nos ajudar a otimizar o projeto. Na Seção 17.4, por exemplo, usaremos o método do potencial para analisar uma tabela que se expande e se contrai dinamicamente.

17.1 ANÁLISE AGREGADA

Em ***análise agregada***, mostramos que, para todo n, uma sequência de n operações demora o tempo do *pior caso* $T(n)$ no total. Portanto, no pior caso, o custo médio, ou ***custo amortizado***, por operação é $T(n)/n$. Observe que esse custo amortizado se aplica a cada operação, mesmo quando há vários tipos de operações na sequência. Os outros dois métodos que estudaremos neste capítulo, o método de contabilidade e o método do potencial, podem atribuir custos amortizados diferentes a tipos de operações diferentes.

Operações de pilha

Em nosso primeiro exemplo de análise agregada, analisamos pilhas que foram aumentadas com uma nova operação. A Seção 10.1 apresentou as duas operações fundamentais de pilha, cada uma das quais demora o tempo $O(1)$:

PUSH(S, x) insere o objeto x sobre a pilha S.

POP(S) retira o topo da pilha S e retorna o objeto retirado da pilha. Chamar POP em uma pilha vazia gera um erro.

Visto que cada uma dessas operações é executada no tempo $O(1)$, vamos considerar o custo de cada uma igual a 1. Então, o custo total de uma sequência de n operações PUSH e POP é n e, por consequência, o tempo de execução real para n operações é Q(n).

Agora acrescentamos a operação de pilha MULTIPOP(S, k) que remove os k objetos no topo da pilha S ou a pilha inteira se ela contiver menos de k objetos. Claro que tomaremos k positivo; senão, a operação MULTIPOP deixaria a pilha como está. No pseudocódigo a seguir, a operação STACK-EMPTY retorna TRUE se não há nenhum objeto na pilha no momento considerado, caso contrário retorna FALSE.

MULTIPOP(S,k)
1 **while** not STACK-EMPTY(S) e $k > 0$
2 POP(S)
3 $k = k - 1$

A Figura 17.1 mostra um exemplo de MULTIPOP.

Qual é o tempo de execução de MULTIPOP(S, k) em uma pilha de s objetos? O tempo de execução real é linear em relação ao número de operações POP realmente executadas e, assim, podemos analisar MULTIPOP considerando o custo abstrato de 1 para cada uma das operações PUSH e POP. O número de iterações do laço **while** é o número $\min(s, k)$ de objetos retirados da pilha. Cada iteração do laço faz uma chamada a POP na linha 2. Assim, o custo total de MULTIPOP é $\min(s, k)$ e o tempo de execução real é uma função linear desse custo.

Vamos analisar uma sequência de n operações PUSH, POP e MULTIPOP em uma pilha inicialmente vazia. O custo do pior caso de uma operação MULTIPOP na sequência é $O(n)$, já que o tamanho da pilha é no máximo n. Portanto, o tempo do pior caso de qualquer operação de pilha é $O(n)$ e, consequentemente, uma sequência de n operações custa $O(n_2)$, já que podemos ter $O(n)$ operações MULTIPOP custando $O(n)$ cada uma. Embora essa análise esteja correta, o resultado $O(n_2)$ que obtivemos considerando o custo do pior caso de cada operação individual não é preciso.

---

topo → 23
17
6
39
10
47
(a)

topo → 10
47
(b)

(c)

Figura 17.1 A ação de MULTIPOP em uma pilha S, mostrada inicialmente em (**a**). Os quatro objetos no topo são retirados por MULTIPOP(S, 4), e o resultado é mostrado em (**b**). A próxima operação é MULTIPOP(S, 7), que esvazia a pilha — mostrada em (**c**) —, já que restam menos de sete objetos.

Usando análise agregada, podemos obter um limite superior melhor que considera a sequência inteira de n operações. De fato, embora uma única operação MULTIPOP possa ser custosa, qualquer sequência de n operações PUSH, POP e MULTIPOP em uma pilha inicialmente vazia custará no máximo $O(n)$. Por quê? Podemos retirar cada objeto de uma pilha no máximo uma vez para cada vez que o inserimos na pilha. Portanto, o número de vezes que POP pode ser chamada em uma pilha não vazia, incluídas as chamadas dentro de MULTIPOP, é no máximo o número de operações PUSH, que é no máximo n. Para qualquer valor de n, qualquer sequência de n operações PUSH, POP e MULTIPOP demora o tempo total $O(n)$. O custo médio de uma operação é $O(n)/n = O(1)$. Em análise agregada, o custo amortizado atribuído a cada operação é o custo médio. Portanto, nesse exemplo todas as três operações de pilha têm um custo amortizado de $O(1)$.

Destacamos novamente que, embora tenhamos acabado de mostrar que o custo médio, e consequentemente o tempo de execução, de uma operação de pilha é $O(1)$, não usamos raciocínio probabilístico. Na realidade, mostramos um limite de *pior caso* $O(n)$ em uma sequência de n operações. Dividindo esse custo total por n temos o custo médio por operação, ou o custo amortizado.

Incrementar um contador binário

Como outro exemplo de análise agregada, considere o problema de implementar um contador binário de k bits que efetua contagem crescente a partir de 0. Usamos como contador um arranjo $A[0 \,..\, k-1]$ de bits, onde $A.comprimento = k$. Um número binário x que é armazenado no contador tem seu bit de ordem mais baixa em $A[0]$ e seu bit de ordem mais alta em $A[k-1]$, de modo que $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Inicialmente $x = 0$ e assim $A[i] = 0$ para $i = 0\ 1 \ldots k-1$. Para somar 1 (módulo 2^k) ao valor do contador, utilizamos o seguinte procedimento:

```
INCREMENT(A)
1   i = 0
2   while i < A.comprimento e A[i] = 1
3       A[i] = 0
4       i = i + 1
5   if i < A.comprimento
6       A[i] = 1
```

A Figura 17.2 mostra o que acontece a um contador binário quando o incrementamos 16 vezes, começando com o valor inicial 0 e terminando com o valor 16. No início de cada iteração do laço **while** nas linhas 2−4, desejamos adicionar um 1 na posição i. Se $A[i] = 1$, então a adição de 1 inverte o bit para 0 na posição i e produz vai-um de 1, que será somado na posição $i + 1$ na próxima iteração do laço. Caso contrário, o laço termina e então, se $i < k$, sabemos que $A[i] = 0$, de modo que a linha 6 adiciona um 1 na posição i, invertendo o bit de 0 para 1. O custo de cada operação INCREMENT é linear em relação ao número de bits invertidos.

Como ocorre no exemplo da pilha, uma análise superficial produz um limite que é correto mas não preciso. Uma única execução de INCREMENT demora o tempo $Q(k)$ no pior caso, no qual o arranjo A contém somente 1s. Portanto, uma sequência de n operações INCREMENT em um contador inicialmente igual a zero demora o tempo $O(nk)$ no pior caso.

Podemos restringir nossa análise para produzir um custo de pior caso $O(n)$ para uma sequência de n operações INCREMENT observando que nem todos os bits são invertidos toda vez que INCREMENT é chamada. Como mostra a Figura 17.2, $A[0]$ é invertido toda vez que INCREMENT é chamada. O bit de ordem mais alta seguinte, $A[1]$, só é invertido em vezes alternadas: uma sequência de n operações INCREMENT em um contador que é inicialmente zero faz $A[1]$ inverter

$n/2$ vezes. De modo semelhante, o bit $A[2]$ é invertido somente de quatro em quatro vezes ou $n/4$ vezes em uma sequência de n operações INCREMENT. Em geral, para $i = 0, 1, \ldots, k - 1$, o bit $A[i]$ é invertido $n/2^i$ vezes em uma sequência de n operações INCREMENT em um contador que inicialmente é zero. Para $i \geq k$, o bit $A[i]$ não existe, portanto não pode ser invertido. Assim, o número total de inversões na sequência é

Valor do contador	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Custo total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figura 17.2 Um contador binário de 8 bits à medida que seu valor vai de 0 a 16 por uma sequência de 16 operações INCREMENT. Bits que são invertidos para chegar ao próximo valor estão sombreados. O custo de execução para a inversão de bits é mostrado à direita. Observe que o custo total é sempre menor que duas vezes o número total de operações INCREMENT.

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n\sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

pela equação (A.6). Portanto, o tempo do pior caso para uma sequência de n operações INCRE-MENT em um contador inicialmente igual a zero é $O(n)$. O custo médio de cada operação e, portanto, o custo amortizado por operação, é $O(n)/n = O(1)$.

Exercícios

17.1-1 Se o conjunto de operações de pilha incluísse uma operação MULTIPUSH, que introduz k itens na pilha, o limite $O(1)$ para o custo amortizado de operações de pilha continuaria válido?

17.1-2 Mostre que, se uma operação DECREMENT fosse incluída no exemplo do contador de k bits, n operações poderiam custar até o tempo Q(nk).

17.1-3 Suponha que executamos uma sequência de n operações em uma estrutura de dados na qual a i-ésima operação custa i, se i é uma potência exata de 2, e 1 em caso contrário. Utilize a análise agregada para determinar o custo amortizado por operação.

17.2 O MÉTODO DE CONTABILIDADE

No ***método da contabilidade*** de análise amortizada, atribuímos cobranças diferentes a operações diferentes, sendo que algumas operações são cobradas a mais ou a menos do que realmente custam. Denominamos ***custo amortizado*** o valor que cobramos por uma operação**.** Quando o custo amortizado de uma operação excede seu custo real, atribuímos a diferença a objetos específicos na estrutura de dados como ***crédito***. Mais adiante, o crédito pode ajudar a pagar operações posteriores cujo custo amortizado é menor que seu custo real. Assim, podemos considerar o custo amortizado de uma operação como repartido entre seu custo real e o crédito que é depositado ou consumido. Esse método é muito diferente da análise agregada, na qual todas as operações têm o mesmo custo amortizado.

Devemos escolher os custos amortizados de operações cuidadosamente. Se quisermos mostrar que, no pior caso, o custo médio por operação é pequeno por análise com custos amortizados, temos de assegurar que o custo amortizado total de uma sequência de operações dá um limite superior para o custo real total da sequência. Além disso, como ocorre na análise agregada, essa relação deve se manter válida para todas as sequências de operações. Se denotarmos o custo real da i-ésima operação por c_i e o custo amortizado da i-ésima operação por $\hat{c}_i$, exigiremos

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \tag{17.1}$$

para todas as sequências de n operações. O crédito total armazenado na estrutura de dados é a diferença entre o custo amortizado total e o custo real total ou $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$. Pela desigualdade (17.1), o crédito total associado à estrutura de dados deve ser não negativo em todos os momentos. Se alguma vez permitíssemos que o crédito total se tornasse negativo (o resultado de cobrar a menos por operações anteriores com a promessa de reembolsar a quantia mais tarde), então os custos amortizados totais incorridos naquele momento estariam abaixo dos custos reais totais incorridos; para a sequência de operações até aquele mesmo momento, o custo amortizado total não seria um limite superior para custo real total. Assim, devemos cuidar para que o crédito total na estrutura de dados nunca se torne negativo.

Operações de pilha

Para ilustrar o método de contabilidade de análise amortizada, vamos voltar ao exemplo da pilha. Lembre-se de que os custos reais das operações eram

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min(k, s)$,

onde k é o argumento fornecido a MULTIPOP e s é o tamanho da pilha quando ela é chamada. Vamos atribuir os seguintes custos amortizados:

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

Observe que o custo amortizado de MULTIPOP é uma constante (0), enquanto o custo real é variável. Aqui, todos os três custos amortizados são constantes. Em geral, os custos amortizados das operações sob consideração podem ser diferentes um do outro e até mesmo assintoticamente diferentes.

Agora mostraremos que é possível pagar qualquer sequência de operações de pilha debitando dos custos amortizados. Suponha que usamos uma nota de um real para representar cada unidade de custo. Começamos com uma pilha vazia. Lembre-se da analogia da Seção 10.1 entre a estrutura de dados de pilha e uma pilha de pratos em um restaurante. Quando introduzimos um prato na pilha, usamos 1 real para pagar o custo propriamente dito do empilhamento e ficamos com um crédito de 1 real (além dos 2 reais cobrados), que deixamos em cima do prato. Em qualquer instante, há um real de crédito em cima de cada prato na pilha.

O real guardado no prato serve como pagamento prévio do custo de retirá-lo da pilha. Quando executamos uma operação POP, não cobramos nada por ela e pagamos seu custo real usando o crédito armazenado na pilha. Para retirar um prato da pilha, tomamos o real de crédito do prato e o utilizamos para pagar o custo real da operação. Assim, cobrando um pouco mais pela operação PUSH, não precisamos cobrar nada pela operação POP.

Além disso, também não precisamos cobrar nada por operações MULTIPOP. Para retirar o primeiro prato da pilha, tomamos o real de crédito do prato e o utilizamos para pagar o custo propriamente dito de uma operação POP. Para retirar um segundo prato, temos novamente um real de crédito no prato para pagar a operação POP, e assim sucessivamente. Então, sempre cobramos antecipadamente o suficiente para pagar operações MULTIPOP. Em outras palavras, visto que cada prato na pilha tem 1 real de crédito e a pilha sempre tem um número não negativo de pratos, asseguramos que a quantia que temos de crédito é sempre não negativa. Assim, para *qualquer* sequência de n operações PUSH, POP e MULTIPOP, o custo amortizado total é um limite superior para o custo real total. Visto que o custo amortizado total é $O(n)$, também é esse o custo real total.

Incrementar um contador binário

Como outra ilustração do método de contabilidade, analisamos a operação INCREMENT em um contador binário que começa em zero. Conforme observamos antes, o tempo de execução dessa operação é proporcional ao número de bits invertidos, que usaremos como nosso custo para esse exemplo. Vamos utilizar uma vez mais uma nota de 1 real para representar cada unidade de custo (a inversão de um bit nesse exemplo).

No caso da análise amortizada, vamos cobrar um custo amortizado de 2 reais para atribuir um bit com 1 (ligar). Quando um bit é ligado, usamos 1 real (dos 2 reais cobrados) para pagar a própria configuração do bit e colocamos o outro real no bit como crédito para ser usado mais tarde quando convertermos o bit de volta para 0. Em qualquer

instante, todo 1 no contador tem um real de crédito e, assim, não precisamos cobrar nada para desligar um bit apenas pagamos o desligamento com a nota de real no bit.

Agora podemos determinar o custo amortizado de INCREMENT. O custo de desligar os bits dentro do laço **while** é pago pelos reais nos bits que são desligados. O procedimento INCREMENT liga no máximo um bit, na linha 6 e, portanto, o custo amortizado de uma operação INCREMENT é no máximo 2 reais. A quantidade de 1s no contador nunca se torna negativa e, portanto, a quantia de crédito permanece não negativa o tempo todo. Assim, para n operações INCREMENT, o custo amortizado total é $O(n)$, o que limita o custo real total.

Exercícios

17.2-1 Suponha que executamos uma sequência de operações de pilha em uma pilha cujo tamanho nunca excede k. Após cada k operações, fazemos uma cópia da pilha inteira como backup. Mostre que o custo de n operações de pilha, incluindo copiar a pilha, é $O(n)$ atribuindo custos amortizados adequados às várias operações de pilha.

17.2-2 Faça novamente o Exercício 17.1-3, usando um método de análise da contabilidade.

17.2-3 Suponha que desejamos não apenas incrementar um contador, mas também reinicializá-lo em zero (isto é, fazer com que todos os seus bits sejam 0). Contando como Q(1), o tempo para examinar ou modificar um bit, mostre como implementar um contador como um arranjo de bits de modo tal que qualquer sequência de n operações INCREMENT e RESET demore o tempo $O(n)$ em um contador inicialmente em zero. (*Sugestão*: Mantenha um ponteiro para o valor 1 de ordem mais alta.)

17.3 O MÉTODO DO POTENCIAL

Em vez de representar o trabalho pago antecipadamente como crédito armazenado por objetos específicos na estrutura de dados, o ***método do potencial*** de análise amortizada representa o trabalho pago antecipadamente como "energia potencial" ou apenas "potencial", que pode ser liberado para pagamento de operações futuras. Associamos o potencial à estrutura de dados como um todo, em vez de associá-lo a objetos específicos dentro da estrutura de dados.

O método do potencial funciona da maneira descrita a seguir. Executaremos n operações, começando com uma estrutura de dados inicial D_0. Para cada $i = 1, 2, ..., n$, seja c_i o custo real da i-ésima operação e seja D_i a estrutura de dados que resulta após a aplicação da i-ésima operação à estrutura de dados D_i-1. Uma ***função potencial*** mapeia cada estrutura de dados D_i para um número real (D_i), que é o ***potencial*** associado à estrutura de dados D_i. O ***custo amortizado*** da i-ésima operação referente à função potencial é definido por

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).. \tag{17.2}$$

Então, o custo amortizado de cada operação é seu custo real mais a mudança no potencial devido à operação. Pela equação (17.2), o custo amortizado total das n operações é

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \end{aligned} \tag{17.3}$$

A segunda igualdade decorre da equação (A.9) porque os termos (D_i) se cancelam.

Se pudermos definir uma função potencial   de tal modo que $(D_n) \geq (D_0)$, então o custo amortizado total $\sum_{i=1}^{n} \hat{c}_i$ dá um limite superior para o custo real total $\sum_{i=1}^{n} c_i$. Na prática, nem sempre sabemos quantas operações poderiam ser executadas. Portanto, se exigimos que $(D_i) \geq (D_0)$ para todo i, então garantimos, como no método da contabilidade, que pagamos antecipadamente. Normalmente apenas definimos (D_0) como 0 e então mostramos que $(D_i) \geq 0$ para todo i. (Veja no Exercício 17.3-1 um modo fácil para tratar os casos nos quais $(D_0) \neq 0$.)

Intuitivamente, se a diferença de potencial $(D_i) - (D_i{-}1)$ da i-ésima operação é positiva, o custo amortizado representa uma cobrança a mais para a i-ésima operação, e o potencial da estrutura de dados aumenta. Se a diferença de potencial é negativa, o custo amortizado representa uma cobrança a menos para a i-ésima operação, e a redução no potencial paga o custo real da operação.

Os custos amortizados definidos pelas equações (17.2) e (17.3) dependem da escolha da função potencial . Diferentes funções potenciais podem produzir custos amortizados diferentes e ainda assim serem limites superiores para os custos reais. Muitas vezes constatamos que podemos fazer permutas quando escolhemos uma função potencial; a melhor função potencial a utilizar depende dos limites de tempo desejados.

Operações de pilha

Para ilustrar o método do potencial, retornamos mais uma vez ao exemplo das operações de pilha Push, Pop e Multipop. Definimos a função potencial   em uma pilha como o número de objetos na pilha. Para a pilha vazia D_0 com a qual começamos, temos $(D_0) = 0$. Como o número de objetos na pilha nunca é negativo, a pilha D_i que resulta após a i-ésima operação tem potencial não negativo e, assim,

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0)\,. \end{aligned}$$

Portanto, o custo amortizado total de n operações em relação a   representa um limite superior para o custo real.

Agora vamos calcular os custos amortizados das várias operações de pilha. Se a i-ésima operação em uma pilha que contém s objetos é uma operação Push, a diferença de potencial é

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1\,. \end{aligned}$$

Pela equação (17.2), o custo amortizado dessa operação Push é

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1+1 \\ &= 2. \end{aligned}$$

Suponha que a i-ésima operação na pilha seja Multipo(S, k), o que resulta na retirada de $k' = \min(k, s)$ objetos da pilha. O custo real da operação é k', e a diferença de potencial é

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$

Assim, o custo amortizado da operação Multipop é

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0. \end{aligned}$$

De modo semelhante, o custo amortizado de uma operação Pop comum é 0.

O custo amortizado de cada uma das três operações é $O(1)$ e, por isso, o custo amortizado total de uma sequência de n operações é $O(n)$. Visto que já demonstramos que $(D_i) \geq (D_0)$, o custo amortizado total de n operações é um limite superior para o custo real total. Portanto, o custo do pior caso de n operações é $O(n)$.

Incrementar um contador binário

Como outro exemplo do método do potencial, examinamos novamente como incrementar um contador binário. Dessa vez, definimos o potencial do contador após a i-ésima operação Increment como b_i, a quantidade de 1s no contador após a i-ésima operação.

Vamos calcular o custo amortizado de uma operação Increment. Suponha que a i-ésima operação Increment modifique t_i bits. Então, o custo real da operação é no máximo $t_i + 1$, já que, além de modificar t_i bits, ela liga no máximo um bit. Se $b_i = 0$, então a i-ésima operação modifica todos os k bits, e $b_i -1 = t_i = k$. Se $b_i > 0$, então $b_i = b_i -1 - t_i + 1$. Em qualquer dos casos, $b_i \leq b_i -1 - t_i + 1$, e a diferença de potencial é

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_{i+1}) - b_{i-1} \\ &= 1 - t_i\,. \end{aligned}$$

Portanto, o custo amortizado é

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2. \end{aligned}$$

Se o contador começa em zero, então $(D_0) = 0$. Visto que $(D_i) \geq 0$ para todo i, o custo amortizado total de uma sequência de n operações Increment é um limite superior para o custo real total e, assim, o custo do pior caso de n operações Increment é $O(n)$.

O método do potencial nos dá um caminho fácil para analisar o contador até mesmo quando ele não começa em zero. O contador começa com uma quantidade b_0 de 1s e, após n operações Increment, ele tem uma quantidade b_n de 1s, onde $0 \leq b_0$, $b_n \leq k$. (Lembre-se de que k é o número de bits no contador.) Podemos reescrever a equação (17.3) como

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0). \tag{17.4}$$

Temos $\hat{c}_i \leq 2$ para todo $1 \leq i \leq n$. Visto que $(D\ ) = b$ e $(D\ ) = b$, o custo real total de n operações Increment é

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0. \end{aligned}$$

Observe em particular que, como $b_0 \leq k$, o custo real total é $O(n)$, contanto que $k = O(n)$. Em outras palavras, se executarmos no mínimo (k) operações INCREMENT, o custo real total será $O(n)$, não importando o valor inicial que o contador contém.

Exercícios

17.3-1 Suponha que tenhamos uma função potencial tal que $(D_i) \geq (D_0)$ para todo i, mas $(D_i) \neq 0$. Mostre que existe uma função potencial ’tal que $'(D_0) = 0$, $'(D_i) \geq 0$ para todo $i \geq 1$ e os custos amortizados se usarmos são iguais aos custos amortizados se usarmos .

17.3-2 Faça novamente o Exercício 17.1-3 usando um método de potencial para a análise.

17.3-3 Considere uma estrutura de dados de heap de mínimo binário comum com n elementos que suporte as instruções INSERT e EXTRACT-MIN no tempo do pior caso $O(\lg n)$. Dê uma função potencial tal que o custo amortizado de INSERT seja $O(\lg n)$ e o custo amortizado de EXTRACT-MIN seja $O(1)$, e mostre que ela funciona.

17.3-4 Qual é o custo total de executar n das operações de pilha PUSH, POP e MULTIPOP considerando que a pilha começa com s_0 objetos e termina com s_n objetos?

17.3-5 Suponha que um contador comece em um número que tem a quantidade b de 1s em sua representação binária, em vez de começar em 0. Mostre que o custo de executar n operações INCREMENT é $O(n)$ se $n = (b)$. (Não considere b constante.)

17.3-6 Mostre como implementar uma fila com duas pilhas comuns (Exercício 10.1-6) de modo que o custo amortizado de cada operação ENQUEUE e de cada operação DEQUEUE seja $O(1)$.

17.3-7 Projete uma estrutura de dados para suportar as duas operações a seguir para um multiconjunto dinâmico S de inteiros que permite valores duplicados:

INSERT(S, x) insere x no conjunto S.

DELETE-LARGER-HALF(S) elimina os $|S|/2$ maiores elementos de S.

Explique como implementar essa estrutura de dados de modo que qualquer sequência de m operações INSERT e DELETE-LARGER-HALF seja executada no tempo $O(m)$. Sua implementação deve também incluir um modo de obter como saída os elementos de S no tempo $O(|S|)$.

17.4 TABELAS DINÂMICAS

Nem sempre sabemos antecipadamente quantos objetos algumas aplicações armazenarão em uma tabela. Poderíamos alocar espaço para uma tabela e só mais tarde constatar que tal espaço não é suficiente. Então, teríamos de realocar a tabela com um tamanho maior e copiar todos os objetos armazenados na tabela original para a nova tabela maior. De modo semelhante, se muitos objetos já foram eliminados da tabela, poderia ser vantajoso realocar tal tabela com um tamanho menor. Nesta seção, estudaremos esse problema de expandir e contrair dinamicamente uma tabela. Usando a análise amortizada, mostraremos que o custo amortizado das operações de inserção e eliminação é apenas $O(1)$, embora o custo real de uma operação seja grande quando ela ativa uma expansão ou uma contração. Além disso, veremos como garantir que o espaço não utilizado em uma tabela dinâmica nunca exceda uma fração constante do espaço total. Supomos que a tabela dinâmica suporta as operações TABLE-INSERT e TABLE-DELETE. TABLE-INSERT insere na tabela um item que ocupa uma única ***posição***, isto é, um espaço para um só item. Do mesmo modo, TABLE-DELETE

elimina um item da tabela e por isso libera uma posição. Os detalhes do método de estruturação de dados usado para organizar a tabela não são importantes; poderíamos usar uma pilha (Seção 10.1), um heap (Capítulo 6) ou uma tabela hash (Capítulo 11). Também poderíamos usar um arranjo ou uma coleção de arranjos para implementar armazenamento de objetos, como fizemos na Seção 10.3.

Veremos que é conveniente utilizar um conceito introduzido em nossa análise do hashing (Capítulo 11). Definimos o ***fator de carga*** $a(T)$ de uma tabela não vazia T como o número de itens armazenados na tabela dividido pelo tamanho (número de posições) da tabela. Atribuímos tamanho 0 a uma tabela vazia (uma tabela sem itens) e definimos seu fator de carga como 1. Se o fator de carga de uma tabela dinâmica é limitado por baixo por uma constante, o espaço não utilizado na tabela nunca é maior que uma fração constante da quantidade total de espaço. Começamos analisando uma tabela dinâmica na qual só inserimos itens. Em seguida consideramos o caso mais geral em que inserimos e eliminamos itens.

17.4.1 EXPANSÃO DE TABELAS

Vamos supor que o armazenamento para uma tabela seja alocado como um arranjo de posições. Uma tabela está cheia quando todas as posições foram usadas ou, o que é equivalente, quando seu fator de carga é 1.[1] Em alguns ambientes de software, se é feita uma tentativa para inserir um item em uma tabela cheia, a única alternativa é abortar a operação com um erro. Porém, levaremos em conta que nosso ambiente de software, como muitos ambientes modernos, fornece um sistema de gerenciamento de memória que pode alocar e liberar blocos de armazenamento por requisição. Assim, ao inserirmos um item em uma tabela cheia, poderemos ***expandi-la*** alocando uma nova tabela com mais posições que a tabela antiga. Como sempre precisamos que a tabela resida em memória contígua, temos de alocar um novo arranjo para a tabela maior e depois copiar itens da tabela antiga para a tabela nova.

Uma heurística comum aloca uma nova tabela que tenha duas vezes o número de posições da antiga. Se as únicas operações de tabela são inserções, então o fator de carga da tabela é sempre no mínimo 1/2 e, assim, a quantidade de espaço desperdiçado nunca excede metade do espaço total na tabela.

No pseudocódigo a seguir, consideramos que T é um objeto que representa a tabela. O atributo $T.tabela$ contém um ponteiro para o bloco de armazenamento que representa a tabela, $T.num$ contém o número de itens na tabela e $T.tamanho$ dá o número total de posições na tabela. Inicialmente, a tabela está vazia: $T.num = T.tamanho = 0$.

```
TABLE-INSERT(T, x)
 1   if T.tamanho == 0
 2         alocar T.tabela com 1 posição
 3         T.tamanho = 1
 4   if T.num == T.tamanho
 5         alocar nova-tabela com 2 × T.tamanho posições
 6         inserir todos os itens em T.tabela em nova-tabela
 7         liberar T.tabela
 8         T.tabela = nova-tabela
 9         T.tamanho = 2 × T.tamanho
10   inserir x em T.tabela
11   T.num = T.num + 1
```

Observe que temos dois procedimentos de "inserção" aqui: o procedimento TABLE-INSERT propriamente dito e a ***inserção elementar*** em uma tabela nas linhas 6 e 10. Podemos analisar o tempo de execução de TABLE-INSERT em termos do número de inserções elementares atribuindo um custo igual a 1 para cada inserção elementar. Supomos que o tempo de execução real de TABLE-INSERT é linear em relação ao tempo gasto para inserir itens individuais, de modo que a sobrecarga de alocação para uma tabela inicial na linha 2 é constante, e a sobrecarga de alocação e liberação de armazenamento nas linhas 5 e 7 é dominada pelo custo de transferir itens na linha 6. Denominamos ***expansão*** o evento no qual as linhas 5−9 são executadas.

Vamos analisar uma sequência de n operações TABLE-INSERT em uma tabela inicialmente vazia. Qual é o custo c_i da i-ésima operação? Se há espaço para o novo item na tabela atual (ou se essa é a primeira operação), então $c_i = 1$, visto que só precisamos executar a única inserção elementar na linha 10. Entretanto, se a tabela atual está cheia e ocorre uma expansão, então $c_i = i$: o custo é 1 para a inserção elementar na linha 10 mais $i - 1$ para os itens que temos de copiar da tabela antiga para a tabela nova na linha 6. Se executarmos n operações, o custo do pior caso de uma operação é $O(n)$, o que acarreta um limite superior de $O(n_2)$ para o tempo total de execução de n operações.

Esse limite não é justo porque raramente expandimos a tabela no curso de n operações TABLE-INSERT. Especificamente, a i-ésima operação provoca uma expansão somente quando $i - 1$ é uma potência exata de 2. O custo amortizado de uma operação é de fato $O(1)$, como podemos mostrar usando análise agregada. O custo da i-ésima operação é

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$$

Portanto, o custo total de n operações TABLE-INSERT é

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n, \end{aligned}$$

já que no máximo n operações custam 1 e os custos das operações restantes formam uma série geométrica. Visto que o custo total de n operações TABLE-INSERT é limitado por $3n$, o custo amortizado de uma única operação é, no máximo, 3.

Usando o método de contabilidade, podemos ter uma ideia do motivo por que o custo amortizado de uma operação TABLE-INSERT deve ser 3. Intuitivamente, cada item paga três inserções elementares: a sua própria inserção na tabela atual, a sua movimentação quando a tabela é expandida e a modificação de um outro item que já tinha sido movido uma vez quando a tabela foi expandida. Por exemplo, suponha que o tamanho da tabela seja m imediatamente após uma expansão. Então, a tabela contém $m/2$ itens e não dispõe de nenhum crédito. Cobramos 3 reais para cada inserção. A inserção elementar que ocorre imediatamente custa 1 real. Colocamos um outro real como crédito no item inserido. Colocamos o terceiro real como crédito em um dos $m/2$ itens que já estão na tabela. A tabela só estará cheia novamente quando inserirmos outros $m/2 - 1$ itens; assim, quando a tabela contiver m itens e estiver cheia, teremos colocado um real em cada item para pagar pela sua reinserção durante a expansão.

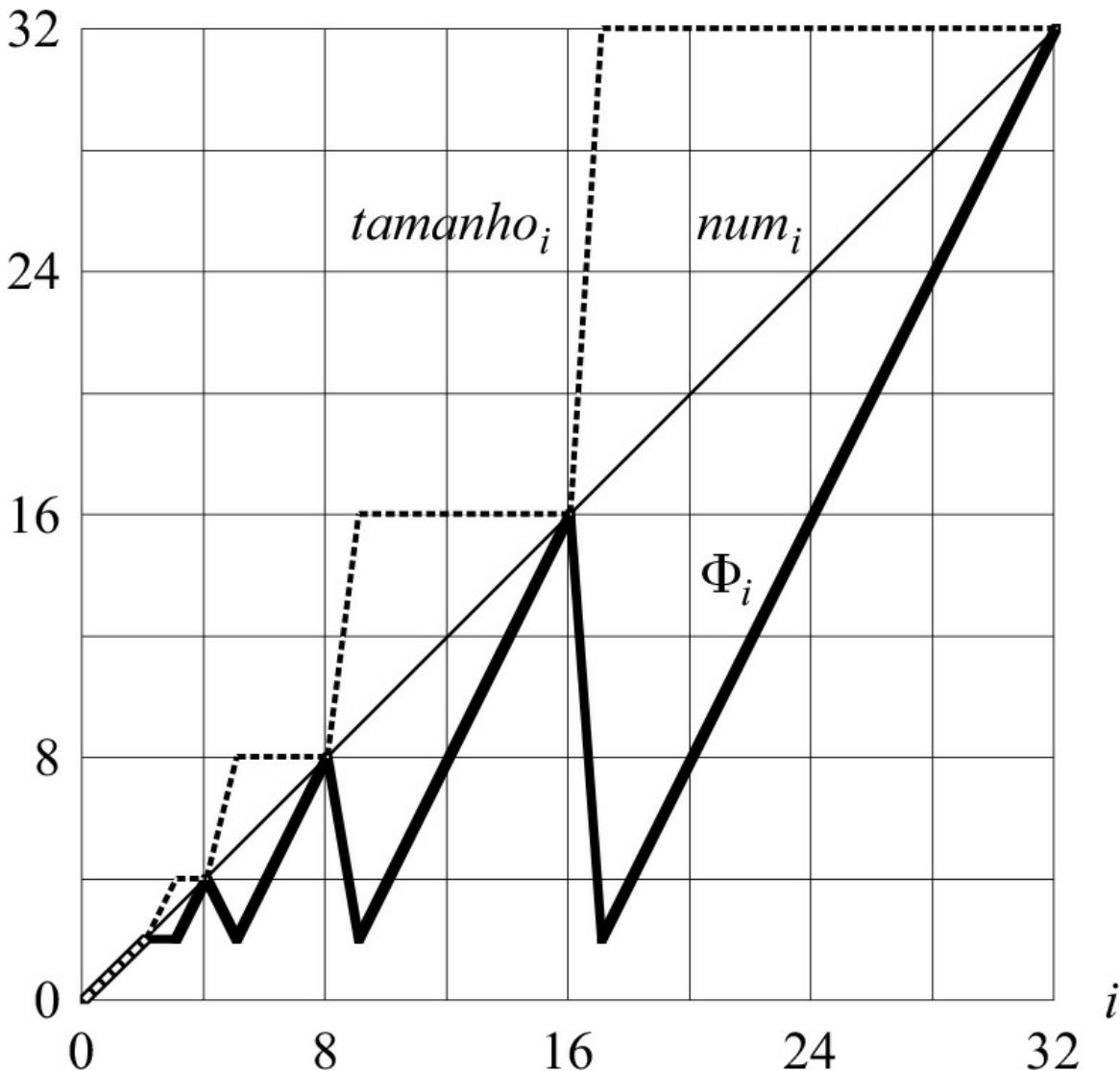
Podemos usar o método do potencial para analisar uma sequência de n operações TABLE-INSERT e o usaremos na Seção 17.4.2 para projetar uma operação TABLE-DELETE, que também tem custo amortizado $O(1)$. Começamos definindo uma função potencial que é igual a 0 imediatamente após uma expansão, mas que aumenta até chegar no tamanho da tabela no momento em que ela está cheia, de modo que podemos pagar a próxima expansão com o potencial. A função

$$\Phi(T) = 2 \cdot T.num - T.tamanho \tag{17.5}$$

é uma possibilidade. Imediatamente após uma expansão, temos $T.num = T.tamanho/2$ e, assim, $(T) = 0$, como desejado. Imediatamente antes de uma expansão, temos $T.num - T.tamanho$ e, assim, $(T) = T.num$, como desejado. O valor inicial do potencial é 0 e, visto que a tabela está sempre no mínimo metade cheia, $T.num \geq T.tamanho/2$, o que

implica que (T) é sempre não negativo. Assim, a soma dos custos amortizados de n operações TABLE-INSERTE é um limite superior para a soma dos custos reais.

Para analisar o custo amortizado da i-ésima operação TABLE-INSERT, representamos por num_i o número de itens armazenados na tabela após a i-ésima operação, por $tamanho_i$ o tamanho total da tabela após a i-ésima operação, e por ^{i} o potencial após a i-ésima operação. Inicialmente, temos $num_0 = 0$, $tamanho_0 = 0$ e $^{0} = 0$.

Se a i-ésima operação TABLE-INSERTE não ativa uma expansão, temos $tamanho^{i} = tamanho_i - ^{1}$ e o custo amortizado da operação é

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \times num_i - tamanho_i) - (2 \times num_{i-1} - tamanho_{i-1}) \\
&= 1 + (2 \times num_i - tamanho_i) - (2(num_i - 1) - tamanho_i) \\
&= 3 .
\end{aligned}
$$

Se a i-ésima operação ativa uma expansão, temos $tamanho_i = 2 \times tamanho_i{-}^{1}$ e $tamanho_i{-}^{1} =$ $num_i{-}^{1} = num_i - 1$, o que implica $tamanho_i = 2 \times (num_i - 1)$. Assim, o custo amortizado da operação é

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \times num_i - tamanho_i) - (2 \times num_{i-1} - tamanho_{i-1}) \\
&= num_i + (2 \times num_i - 2 \times num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3 .
\end{aligned}
$$

A Figura 17.3 mostra a representação gráfica dos valores de num_i, $tamanho_i$ e ^{i} em relação a i. Observe como o potencial aumenta para pagar a expansão da tabela.

Figura 17.3 Efeito de uma sequência de n operações TABLE-INSERT sobre o número num_i de itens na tabela, o número $tamanho_i$ de posições na tabela e o potencial $_i = 2 \times num_i - tamanho_i$, cada um medido após a i-ésima operação. A linha fina mostra num_i, a linha tracejada mostra $tamanho_i$ e a linha grossa mostra $_i$. Observe que, imediatamente antes de uma expansão, o potencial cresceu até o número de itens na tabela e, assim, ele pode pagar a movimentação de todos os itens para a nova tabela. Depois, o potencial cai até 0, mas é imediatamente aumentado de 2 quando da inserção do item que causou a expansão.

17.4.2 EXPANSÃO E CONTRAÇÃO DE TABELAS

Para implementar uma operação TABLE-DELETE, é bastante simples remover o item especificado da tabela. Porém, para limitar a quantidade de espaço desperdiçado, seria interessante ***contrair*** a tabela quando o fator de carga da tabela se tornar demasiadamente pequeno. A contração de tabelas é análoga à expansão: quando o número de itens na tabela fica muito baixo, alocamos uma nova tabela menor e, em seguida, copiamos os itens da tabela antiga na tabela nova. Então podemos liberar o armazenamento usado para a tabela antiga devolvendo-o ao sistema de gerenciamento de memória. No caso ideal, devemos preservar duas propriedades:

- o fator de carga da tabela dinâmica é limitado por baixo por uma constante.
- o custo amortizado de uma operação de tabela é limitado por cima por uma constante.

Consideramos o custo medido em termos de inserções e eliminações elementares.

Seria natural pensar que devemos dobrar o tamanho da tabela quando um item é inserido em uma tabela cheia e reduzi-lo à metade quando a eliminação de um item resulta em uma tabela menos da metade cheia. Essa estratégia garantiria que o fator de carga da tabela nunca cairia abaixo de 1/2 mas, infelizmente, pode resultar em custo amortizado por operação muito grande. Considere o cenário a seguir. Executamos n operações em uma tabela T, onde n é uma potência exata de 2. As primeiras $n/2$ operações são inserções que, de acordo com nossa análise anterior, têm custo total Q(n). Ao final dessa sequência de inserções, $T.num = T.tamanho = n/2$. Para a segunda série de $n/2$ operações, executamos a seguinte sequência:

inserir, eliminar, eliminar, inserir, inserir, eliminar, eliminar, inserir, inserir, ...

A primeira inserção causa uma expansão da tabela até o tamanho n. As duas eliminações seguintes provocam uma contração da tabela de volta ao tamanho $n/2$. Duas inserções adicionais provocam outra expansão, e assim por diante. O custo de cada expansão e contração é Q(n), e há Q(n) dessas operações. Assim, o custo total das n operações é Q(n_2), e o custo amortizado de uma operação é Q(n).

A desvantagem dessa estratégia é óbvia: após expandirmos a tabela, não eliminamos itens suficientes para pagar uma contração. Do mesmo modo, após contrairmos a tabela, não inserimos itens suficientes para pagar uma expansão.

Podemos aperfeiçoar essa estratégia permitindo que o fator de carga da tabela caia abaixo de 1/2. Especificamente, continuamos a duplicar o tamanho da tabela quando um item é inserido em uma tabela cheia, mas o reduzimos à metade quando a eliminação de um item resulta em uma tabela menos de 1/4 cheia, em vez de 1/2 cheia como antes. Então, o fator de carga da tabela tem um limite inferior dado pela constante 1/4.

Intuitivamente, consideramos que um fator de carga de 1/2 seja ideal e o potencial da tabela seria 0. À medida que o fator de carga se desvia de 1/2 , o potencial cresce, de modo que, ao tempo em que expandimos ou contraímos a tabela, ela já acumulou potencial suficiente para pagar a cópia de todos os itens para a tabela recém-alocada. Assim, precisaremos de uma função potencial que cresceu até T.*num* ao tempo em que o fator de carga tiver crescido até 1 ou diminuído até 1/4. Após expandir ou contrair a tabela, o fator de carga volta a 1/2 e o potencial da tabela diminui e volta a 0.

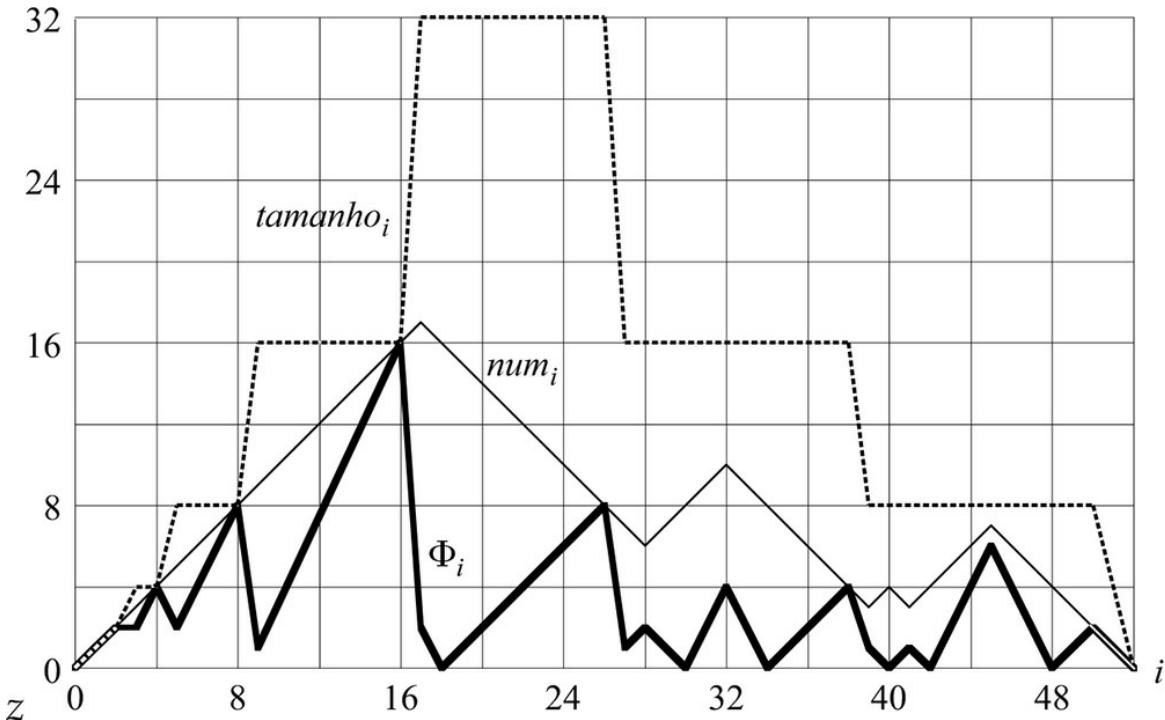
Omitimos o código para TABLEDELETE por ser análogo a TABLE-INSERT. Para a nossa análise, consideraremos que, sempre que o número de itens na tabela cair a 0, liberamos o armazenamento para a tabela. Isto é, se $T.num = 0$, então $T.tamanho = 0$.

Agora podemos usar o método do potencial para analisar o custo de uma sequência de n operações TABLE-INSERT e TABLE-DELETE. Começamos definindo uma função potencial que é 0 imediatamente após uma expansão ou contração e aumenta à medida que o fator de carga aumenta até 1 ou diminui até 1/4. Vamos denotar o fator de carga de uma tabela não vazia T por $a(T) = T.num/T.tamanho$. Visto que para uma tabela vazia $T.num = T.tamanho = 0$ e $a[T] = 1$, sempre temos $T.num = a(T) \times T.tamanho$, quer a tabela esteja vazia ou não. Utilizaremos como nossa função potencial

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.tamanho & \text{se } \alpha(T) \geq 1/2, \\ T.tamanho/2 - Tnum & \text{se } \alpha(T) < 1/2. \end{cases} \quad (17.6)$$

Observe que o potencial de uma tabela vazia é 0 e que o potencial nunca é negativo. Portanto, o custo amortizado total de uma sequência de operações em relação a dá um limite superior para o custo real da sequência.

Antes de continuar com uma análise precisa, fazemos uma pausa para examinar algumas propriedades da função potencial. Observe que, quando o fator de carga é 1/2, o potencial é 0.

Quando o fator de carga é 1, temos $T.tamanho = T.num$, o que implica $= T.num$ e, assim, o potencial pode pagar uma expansão se um item é inserido. Quando o fator de carga é 1/4, temos $T.tamanho = 4 \times T.num$, o que implica $=$ $T.num$ e, assim, o potencial pode pagar uma contração se um item é eliminado.

Para analisar uma sequência de n operações TABLE-INSERT e TABLE-DELETE, sejam c_i o custo real da i-ésima operação, $\hat{c}_i$ seu custo amortizado em relação a , num_i o número de itens armazenados na tabela após a i-ésima operação,

tamanho$_i$ o tamanho total da tabela após a *i*-ésima operação, a^i o fator de carga da tabela após a *i*-ésima operação e ^{i} o potencial após a *i*-ésima operação. Inicialmente, $num_0 = 0$, $tamanho_0 = 0$, $a^0 = 1$ e $^0 = 0$.

Começamos com o caso no qual a *i*-ésima operação é TABLE-INSERT. A análise é idêntica à da expansão de tabela da Seção 17.4.1 se $a^i - {}^1 \geq 1/2$. Quer a tabela seja expandida ou não, o custo amortizado $\hat{c}_i$ da operação é, no máximo, 3. Se $a^{i-1} < 1/2$, a tabela não pode se expandir como esultado da operação, já que ela se expande somente quando $a^i - {}^1 = 1$. Se $a^i < 1/2$, então o custo amortizado da *i*-ésima operação será

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (tamanho_i/2 - num_i) - (tamanho_{i-1}/2 - num_{i-1}) \\
&= 1 + (tamanho_i/2 - num_i) - (tamanho_i/2 - (num_i - 1)) \\
&= 0\,.
\end{aligned}
$$

Figura 17.4 Efeito de uma sequência de *n* operações TABLE-INSERT e TABLE-DELETE sobre o número num_i de itens na tabela, o número $tamanho_i$ de posições na tabela e o potencial

$$
\Phi_i = \begin{cases} 2 \cdot num_i - tamanho_i & \text{se } \alpha_i \geq 1/2, \\ tamanho_i / 2 - num_i & \text{se } \alpha_i < 1/2. \end{cases}
$$

cada um medido após a *i*-ésima operação. A linha fina mostra num_i, a linha tracejada mostra tamanho$_i$, e a linha grossa mostra $_i$. Observe que, imediatamente antes de uma expansão, o potencial aumentou até o número de itens na tabela e, portanto, pode pagar a movimentação de todos os itens para a nova tabela. Da mesma forma, imediatamente antes de uma contração, o potencial aumentou até o número de itens na tabela.

Se $\alpha_{i-1} < 1/2$, mas $\alpha_i \geq 1/2$, então

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi i - \Phi_{i-1} \\
&= 1 + (2 \times num_i - tamanho_i) - (tamanho_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - tamanho_{i-1}) - (tamanho_{i-1}/2 - num_{i-1}) \\
&= 3 \times num_{i-1} - \frac{3}{2} tamanho_{i-1} + 3 \\
&= 3\alpha_{i-1} tamanho_{i-1} - \frac{3}{2} tamanho_{i-1} + 3 \\
&< \frac{3}{2} tamanho_{i-1} - \frac{3}{2} tamanho_{i-1} + 3 \\
&= 3 .
\end{aligned}$$

Assim, o custo amortizado de uma operação TABLE-INSERT é, no máximo, 3.

Examinaremos agora o caso no qual a i-ésima operação é TABLE-DELETE. Nesse caso, $num_i = num_{i-1} - 1$. Se $a_{i-1} < 1/2$, devemos considerar se a operação provoca ou não contração da tabela. Se não provoca, $tamanho_i = tamanho_{i-1}$, e o custo amortizado da operação é

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (tamanho_i/2 - num_i) - (tamanho_{i-1}/2 - num_{i-1}) \\
&= 1 + (tamanho_i/2 - num_i) - (tamanho_i/2 - (num_i + 1)) \\
&= 2 .
\end{aligned}$$

Se $a_{i-1} < 1/2$ e a i-ésima operação ativa uma contração, o custo real da operação é $c_i = num_i + 1$, visto que eliminamos um item e movemos num_i itens. Temos $tamanho_i/2 = tamanho_{i-1}/4 = num_i + 1$ e o custo amortizado da operação é

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (tamanho_i/2 - num_i) - (tamanho_{i-1}/2 - num_{i-1}) \\
&= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \times num_i + 2) - (num_i + 1)) \\
&= 1 .
\end{aligned}$$

Quando a i-ésima operação é uma TABLE-DELETE e $a_{i-1} \geq 1/2$, o custo amortizado também é limitado por cima por uma constante. A análise fica para o Exercício 17.4-2.

Resumindo, visto que o custo amortizado de cada operação é limitado por cima por uma constante, o tempo real para qualquer sequência de n operações em uma tabela dinâmica é $O(n)$.

Exercícios

17.4-1 Suponha que desejamos implementar uma tabela hash dinâmica de endereço aberto. Por que poderíamos considerar que a tabela está cheia quando seu fator de carga alcança algum valor a que é estritamente menor que 1? Descreva brevemente como fazer a inserção em uma tabela hash dinâmica de endereço aberto funcionar de tal maneira que o valor esperado do custo amortizado por inserção seja $O(1)$. Por que o valor esperado do custo real por inserção não é necessariamente $O(1)$ para todas as inserções?

17.4-2 Mostre que, se $a_{i-1} \geq 1/2$ e a i-ésima operação em uma tabela dinâmica é TABLE-DELETE, o custo amortizado da operação em relação à função potencial (17.6) é limitado por cima por uma constante.

17.4-3 Suponha que, em vez de contrair uma tabela reduzindo seu tamanho à metade quando seu fator de carga cai abaixo de 1/4, nós a contraíssemos multiplicando seu tamanho por 2/3 quando seu fator de carga cai abaixo de 1/3. Usando a função potencial

$(T) = |2 \times T.num - T.tamanho|$,

mostre que o custo amortizado de uma operação TABLE-DELETE que utiliza essa estratégia é limitado por cima por uma constante.

17-1 Contador binário com inversão de bits

O Capítulo 30 examina um importante algoritmo denominado transformação rápida de Fourier (FFT — *fast Fourier transform*). A primeira etapa do algoritmo FFT executa uma ***permutação com reversão de bits*** em um arranjo de entrada $A[0 \,..\, n-1]$ cujo comprimento é $n = 2^k$ para algum inteiro não negativo k. Essa permutação troca entre si elementos cujos índices têm representações binárias que são o reverso uma da outra. Podemos expressar cada índice a como uma sequência de k bits $\langle a_{k-1}, a_{k-2}, ..., a_0 \rangle$, onde $a = \sum_{i=0}^{k-1} a_i 2^i$. Definimos

$$\text{rev}_k(\langle a_{k-1}, a_{k-2}, ..., a_0 \rangle) = \langle a_0, a_1, ..., a_{k-1} \rangle;$$

assim,

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i.$$

Por exemplo, se $n = 16$ (ou, o que é equivalente, $k = 4$), então $\text{rev}_k(3) = 12$, já que a representação de 3 com 4 bits é 0011 que, ao ser invertido, dá 1100, a representação com 4 bits de 12.

a. Dada uma função rev_k que é executada no tempo $\Theta(k)$, escreva um algoritmo para executar a permutação com reversão de bits em um arranjo de comprimento $n = 2^k$ no tempo $O(nk)$.

Podemos usar um algoritmo baseado em uma análise amortizada para melhorar o tempo de execução da permutação com reversão de bits. Mantemos um "contador com bits reversos" e um procedimento BIT-REVERSED-INCREMENT que, dado um valor a do contador com bits reversos produz $\text{rev}_k(\text{rev}_k(a) + 1)$. Por exemplo, se $k = 4$ e o contador com inversão de bits começa em 0, chamadas sucessivas a BIT-REVERSED-INCREMENT produzem a sequência

0000, 1000, 0100, 1100, 0010, 1010, ... = 0, 8, 4, 12, 2, 10, ...

b. Suponha que as palavras em seu computador armazenem valores de k bits e que, no tempo unitário, o computador pode manipular os valores binários com operações tais como deslocamentos para a esquerda ou para a direita de valores arbitrários, AND em relação a bits, OR em relação a bits etc. Descreva uma implementação do procedimento BIT-REVERSED-INCREMENT que permita que a permutação com reversão de bits em um arranjo de n elementos seja executada no tempo total $O(n)$.

c. Suponha que você possa deslocar uma palavra somente um bit para a esquerda ou para a direita em tempo unitário. Ainda é possível implementar uma permutação com reversão de bits no tempo $O(n)$?

17-2 Como tornar dinâmica a busca binária

A busca binária de um arranjo ordenado demora um tempo de busca logarítmico, mas o tempo para inserir um novo elemento é linear em relação ao tamanho do arranjo. Podemos melhorar o tempo para inserção mantendo vários arranjos ordenados. Especificamente, suponha que desejemos suportar SEARCH e INSERT em um conjunto de n elementos. Seja $k = \lg(n + 1)$, e seja $\langle n_{k-1}, n_{k-2}, ..., n_0 \rangle$ a representação binária de n. Temos k arranjos ordenados $A_0, A_1, ..., A_{k-1}$ onde, para $i = 0, 1, ..., k-1$, o comprimento do arranjo A_i é 2 . Cada arranjo está cheio ou vazio, dependendo de $n_i = 1$ ou $n_i = 0$, respectivamente. O número total de elementos contidos em todos os k arranjos é, portanto, $\sum_{i=0}^{k-1} n_i 2^i = n$. Embora cada arranjo individual seja ordenado, não há nenhuma relação particular entre elementos de arranjos diferentes.

a. Descreva como executar a operação SEARCH para essa estrutura de dados. Analise seu tempo de execução do pior caso.

b. Descreva como executar a operação INSERT. Analise seu tempo de execução do pior caso e seu tempo de execução amortizado.

c. Discuta como implementar DELETE.

17-3 Árvores de peso balanceado amortizadas

Considere uma árvore de busca binária comum aumentada pelo acréscimo a cada nó x do atributo $x.tamanho$ que dá o número de chaves armazenadas na subárvore com raiz em x. Seja a uma constante na faixa $1/2 \leq a < 1$. Dizemos que um dado nó x é a **balanceado** se $x.esquerda.tamanho \leq a \cdot x.tamanho$ e $x.direita.tamanho \leq a \cdot x.tamanho$. A árvore como um todo é ***a*-balanceada** se todo nó na árvore é a balanceado. A abordagem amortizada para manter árvores de peso balanceado que apresentamos a seguir foi sugerida por G. Varghese.

a. Uma árvore 1/2-balanceada é, em certo sentido, tão balanceada quanto possível. Dado um nó x em uma árvore de busca binária arbitrária, mostre como reconstruir a subárvore com raiz em x de modo que ela se torne 1/2-balanceada. Seu algoritmo deve ser executado no tempo Q($x.tamanho$) e pode utilizar armazenamento auxiliar $O(x.tamanho)$.

b. Mostre que executar uma busca em uma árvore de busca binária a-balanceada de n nós demora $O(\lg n)$ tempo do pior caso.

Para o restante deste problema, considere que a constante a seja estritamente maior que 1/2. Suponha que implementamos INSERT e DELETE da maneira usual para uma árvore de busca binária de n nós, exceto que, após cada uma dessas operações, se qualquer nó na árvore não for mais a-balanceado, "reconstruímos" a subárvore com raiz no mais alto nó com essa propriedade, de modo tal que ela se torne 1/2-balanceada. Analisaremos esse esquema de reconstrução usando o método do potencial. Para um nó x em uma árvore de busca binária T, definimos

$$\Delta(x) = |x.esquerda.tamanho - x.direita.tamanho| ,$$

e definimos o potencial de T como

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x) ,$$

onde c é uma constante suficientemente grande que depende de a.

c. Demonstre que qualquer árvore de busca binária tem potencial não negativo e que uma árvore 1/2-balanceada tem potencial 0.

d. Suponha que m unidades de potencial possam pagar a reconstrução de uma subárvore de m nós. Que tamanho deve ter c em termos de a, de modo que o tempo amortizado para a reconstrução de uma subárvore que não seja a-balanceada seja $O(1)$?

e. Mostre que a inserção ou a eliminação de um nó de uma árvore a-balanceada de n nós tem o custo de tempo amortizado $O(\lg n)$.

17-4 O custo de reestruturação de árvores vermelho-preto

Há quatro operações básicas em árvores vermelho-preto que executam ***modificações estruturais***: inserções de nós, eliminações de nós, rotações e mudanças de cor. Vimos que RB-INSERT e RB-DELETE utilizam somente $O(1)$ rotações, inserções de nós e eliminações de nós para manter as propriedades vermelho-preto, mas podem fazer um número muito maior de mudanças de cor.

a. Descreva uma árvore vermelho-preto correta com n nós tal que chamar RB-INSERT para adicionar o $(n + 1)$-ésimo nó provoque (lg n) mudanças de cor. Então descreva uma árvore vermelho-preto correta com n nós para a qual chamar RB-DELETE em um nó particular provoca (lg n) mudanças de cor.

Embora o número de mudanças de cor do pior caso por operação possa ser logarítmico, provaremos que qualquer sequência de m operações RB-INSERT e RB-DELETE em uma árvore vermelho-preto inicialmente vazia provoca $O(m)$ modificações estruturais no pior caso. Observe que contamos cada mudança de cor como uma modificação estrutural.

b. Alguns dos casos tratados pelo laço principal do código de RB-INSERT-FIXUP e RB-DELETE-FIXUP são ***terminais***: uma vez encontrados, eles fazem o laço terminar após um número constante de operações adicionais. Para cada um dos casos de RB-INSERT-FIXUP e RB-DELETE-FIXUP, especifique quais são terminais e quais não são. (*Sugestão:* Observe as Figuras 13.5, 13.6 e 13.7.)

Primeiro analisaremos as modificações estruturais quando são executadas somente inserções. Seja T uma árvore vermelho-preto e defina (T) como o número de nós vermelhos em T. Suponha que uma unidade de potencial pode pagar as modificações estruturais executadas por qualquer dos três casos de RB-INSERT-FIXUP.

c. Seja T' o resultado da aplicação do Caso 1 de RB-INSERT-FIXUP a T. Mostre que (T') = (T) − 1.

d. Quando inserimos um nó em uma árvore vermelho-preto utilizando RB-INSERT, podemos desmembrar a operação em três partes. Faça uma lista das modificações estruturais e mudanças de potencial resultantes das linhas 1–16 de RB-INSERT para casos não terminais de RB-INSERT-FIXUP e para casos terminais de RB-INSERT-FIXUP.

e. Usando a parte (d), demonstre que o número amortizado de modificações estruturais executadas por qualquer chamada de RB-INSERT é $O(1)$.

Agora desejamos provar que existem $O(m)$ modificações estruturais quando ocorrem inserções e eliminações. Vamos definir, para cada nó x,

$$w(x) = \begin{cases} 0 & \text{se x é vermelho,} \\ 1 & \text{se x é preto e não tem nenhum filho vermelho,} \\ 0 & \text{se x é preto e tem um filho vermelho,} \\ 2 & \text{se x é preto e tem dois filhos vermelhos.} \end{cases}$$

Agora, redefinimos o potencial de uma árvore vermelho-preto T como

$$\Phi(T) = \sum_{x \in T} w(x),$$

e seja T' a árvore que resulta da aplicação de qualquer caso não terminal de RB-INSERT-FIXUP ou RB-DELETE-FIXUP a T.

f. Mostre que (T') ≤ (T) − 1 para todos os casos não terminais de RB-INSERT-FIXUP. Demonstre que o número amortizado de modificações estruturais executadas por qualquer chamada de RB-INSERT-FIXUP é

$O(1)$.

g. Mostre que $(T') \leq (T) - 1$ para todos os casos não terminais de RB-DELETE-FIXUP. Demonstre que o número amortizado de modificações estruturais executadas por qualquer chamada de RB-DELETE-FIXUP é $O(1)$.

h. Complete a prova de que, no pior caso, qualquer sequência de m operações RB-INSERT e RB-DELETE executa modificações estruturais O (m).

17-5 Análise competitiva de listas auto-organizadas com mova-para-frente

Uma ***lista auto-organizada*** é uma lista ligada de ***n*** elementos, na qual cada elemento tem uma chave exclusiva. Quando procuramos um elemento na lista, recebemos uma chave e queremos encontrar um elemento que tenha essa chave.

A lista auto-organizada tem duas propriedades importantes:

1. Para encontrar um elemento na lista, dada a sua chave, devemos percorrer a lista desde o início até encontrar o elemento que tem a chave dada. Se esse elemento é o k-ésimo elemento desde o início da lista, o custo de encontrar o elemento é k.

2. Podemos reordenar os elementos da lista após qualquer operação, de acordo com uma regra dada com um custo dado. Podemos escolher qualquer heurística que quisermos para decidir como reordenar a lista.

Suponha que começamos com um determinada lista de n elementos e que recebemos uma sequência de acesso $s = \langle s_1, s_2, \ldots, s_m \rangle$ de chaves a encontrar, em ordem. O custo da sequência é a soma dos custos dos acessos individuais na sequência.

Dentre os vários modos possíveis de reordenar a lista após uma operação, este problema focaliza a transposição de elementos adjacentes na lista — permutação de suas posições na lista — com um custo unitário para cada operação de transposição. Você mostrará, por meio de uma função potencial, que determinada heurística para reordenação da lista, denominada mova-para-frente, acarreta um custo total que não é pior que quatro vezes o de qualquer outra heurística para manter a ordem da lista — mesmo que a outra heurística conheça antecipadamente a sequência de acesso! Esse tipo de análise é chamada de **análise competitiva**.

Para uma heurística H e uma determinada ordenação inicial da lista, denote por $C_H(s)$. o custo de acesso da sequência s . Seja m o número de acessos em s.

a. Mostre que, se a heurística H não conhecer antecipadamente a sequência de acesso, o pior caso para H em uma sequência de acesso s é $C_H(s) = (mn)$.

Com a heurística ***mova-para-frente***, imediatamente após procurar um elemento x, passamos x para a primeira posição na lista (isto é, para a frente da lista).

Vamos denotar por $\text{rank}_L(x)$ o posto do elemento x na lista L, isto é, a posição de x na lista L. Por exemplo, se x é o quarto elemento em L, $\text{rank}_L(x) = 4$. Seja c_i o custo de acesso s_i usando a heurística move-to-front, que inclui o custo de encontrar o elemento na lista e o custo de passá-lo para a frente da lista por uma série de transposições de elementos adjacentes na lista.

b. Mostre que, se T_i acessa o elemento x na lista L usando a heurística move-to-front, então $c_i = 2 \cdot \text{rank}_L(x) - 1$.

Agora comparamos mova-para-frente com qualquer outra heurística H que processe uma sequência de acesso de acordo com as duas propriedades já citadas. A heurística H pode transpor elementos na lista do jeito que quiser e poderia até conhecer antecipadamente toda a sequência de acesso.

Seja L_i a lista após acesso s^i usando mova-para-frente e seja L^{*i} a lista após acesso s^i usando heurística H. Denotamos o custo de acesso s^i por c_i para a heurística move-to-front e por $c_*{}^i$ para a heurística H. Suponha que a heurística H execute t_* transposições durante o acesso s^i.

c. Na parte (b), você mostrou que $c_i = 2 \cdot \text{rank}_{L_{i-1}}(x) - 1$. Agora, mostre que $c^{*i} = \text{rank}_{L^*_{i-1}}(x) + t^*_{i*}$

Definimos uma ***inversão*** na lista L_i como um par de elementos y e z tal que y precede z em L_i e z precede y na lista L_i. Suponha que a lista L_i tenha q_i inversões após processar a sequência de acesso $\langle s^1, s^2, \ldots, s^i \rangle$. Então, definimos uma função potencial que mapeia L_i para um número real por $(L_i) = 2q_i$. Por exemplo, se L_i tiver os elementos $\langle e, c, a, d, b \rangle$ e L_* tiver os elementos $\langle c, a, b, d, e \rangle$, L terá cinco inversões $((e, c), (e, a), (e, d), (e, b), (d, b))$ e, portanto, $(L_i) = 10$. Observe que $(L_i) \geq 0$ para todo i e que, se mova-para-frente e heurística H começarem com a mesma lista L_0, $(L_0) = 0$.

d. Mostre que uma transposição aumenta o potencial de 2 ou reduz o potencial de 2.

Suponha que acesso T_i encontre o elemento x. Para entender como o potencial muda devido a T_i, vamos particionar os elementos, exceto x, em quatro conjuntos, dependendo de onde eles estão posicionados nas listas exatamente antes do i-ésimo acesso:

- Conjunto A consiste em elementos que precedem x nas listas L_{i-1} e L^*_{i-1*}.
- Conjunto B consiste em elementos que precedem x em L_{i-1} e vêm depois de x em L^*_{i-1*}.
- Conjunto C consiste em elementos que vêm depois de x em L_{i-1} e precedem x em L^*_{i-1*}.
- Conjunto D consiste em elementos que vêm depois de x nas listas L_{i-1} e L^*_{i-1*}.

e. Mostre que $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$ e $\text{rank}_{L^*}(x) = |A| + |C| + 1$.

f. Mostre que acesso s_i provoca uma mudança no potencial de

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t_i^*),$$

onde, como antes, a heurística H executa t_* transposições durante acesso s_i.

Defina o custo amortizado $\hat{c}_i$ de acesso s_i por $\hat{c}_i = c_i + (L_i) - (L_{i-1})$.

g. Mostre que o custo amortizado $\hat{c}_i$ do acesso s_i é limitado por cima por $4c_{*i}$.

h. Conclua que o custo $C_{\text{MTF}}(s)$ da sequência de acesso s com mova-para-frente é, no máximo, quatro vezes o custo $C_H(s)$ de s com qualquer outra heurística H, supondo que ambas as heurísticas começam com a mesma lista.

NOTAS DO CAPÍTULO

Aho, Hopcroft e Ullman [5] usaram análise agregada para determinar o tempo de execução de operações em uma floresta de conjuntos disjuntos; analisaremos essa estrutura de dados usando o método do potencial no Capítulo 21. Tarjan [331] examina os métodos da contabilidade e de potencial de análise amortizada e apresenta diversas aplicações. Ele atribui o método da contabilidade a vários autores, entre eles M. R. Brown, R. E. Tarjan, S. Huddleston e K. Mehlhorn e o método do potencial a D. D. Sleator. O termo “amortizado” se deve a D. D. Sleator e R. E. Tarjan.

Funções potenciais também são úteis para provar limites inferiores para certos tipos de problemas. Para cada configuração do problema, definimos uma função potencial que mapeia a configuração para um número real. Então, determinamos o potencial $_{inicial}$ da configuração inicial, o potencial $_{final}$ da configuração final e a máxima mudança no potencial D_{max} que se deve aqualquer etapa. Portanto, o número de etapas deve ser, no mínimo, $|_{final} - _{inicial}|/|D_{max}|$. Exemplos de funções potencial para provar limites inferiores para complexidade de E/S aparecem em trabalhos de Cormen, Sundquist e Wisniewski [79]; e Floyd [107] e Aggarwal e Vitter [4]. Krumme, Cybenko e Venkataraman [221] aplicaram funções potenciais para provar limites inferiores para ***fofoca***: comunicar um único item de cada vértice em um grafo a todos os outros vértices.

A heurística move-to-front do Problema 17-5 funciona muito bem na prática. Além do mais, se reconhecermos que, quando encontramos um elemento podemos recortá-lo de sua posição na lista e passá-lo para a frente da lista em tempo constante, podemos mostrar que o custo de mova-para-frente é, no máximo, duas vezes o custo de qualquer outra heurística incluindo, novamente, uma que conheça antecipadamente toda a sequência de acesso.

---

[1]Em algumas situações, como no caso de uma tabela hash de endereço aberto, poderemos querer considerar uma tabela cheia se seu fator de carga for igual a alguma constante estritamente menor que 1 (veja o Exercício 17.4-1).

Parte

ESTRUTURAS DE DADOS AVANÇADAS

INTRODUÇÃO

Esta parte volta ao estudo de estruturas de dados que suportam operações em conjuntos dinâmicos, porém em um nível mais avançado que o da Parte III. Por exemplo, dois dos capítulos fazem uso extensivo das técnicas de análise amortizada que vimos no Capítulo 17.

O Capítulo 18 apresenta as B-árvores, que são árvores de busca balanceadas projetadas especificamente para armazenamento em discos. Visto que discos funcionam muito mais lentamente que memória de acesso aleatório, medimos o desempenho de B-árvores não apenas pela quantidade de tempo de computação que as operações em conjuntos dinâmicos consomem, mas também pela quantidade de acessos a disco que elas executam. Para cada operação de B-árvore, o número de acessos a disco aumenta com a altura da B-árvore, porém operações de B-árvore mantêm a altura baixa.

O Capítulo 19 apresenta uma implementação de um heap intercalável, que dá suporte as operações Insert, Minimum, Extract-Min e Union.[1] A operação Union une, ou intercala, dois heaps. Heaps de Fibonacci — a estrutura de dados apresentada no Capítulo 19 — também dão suporte as operações Delete e Decrease-Key. Usamos limites de tempo amortizados para medir o desempenho de heaps de Fibonacci. As operações Insert, Minimum e Union demoram somente tempo real e amortizado $O(1)$ em heaps de Fibonacci, e as operações Extract-Min e Delete demoram o tempo amortizado $O(\lg n)$. Entretanto, a vantagem mais significativa dos heaps de Fibonacci é que Decrease-Key leva somente o tempo amortizado $O(1)$. Como a operação Decrease-Key demora tempo amortizado constante, heaps de Fibonacci são componentes fundamentais de alguns dos algoritmos assintoticamente mais rápidos existentes até hoje para problemas de grafos.

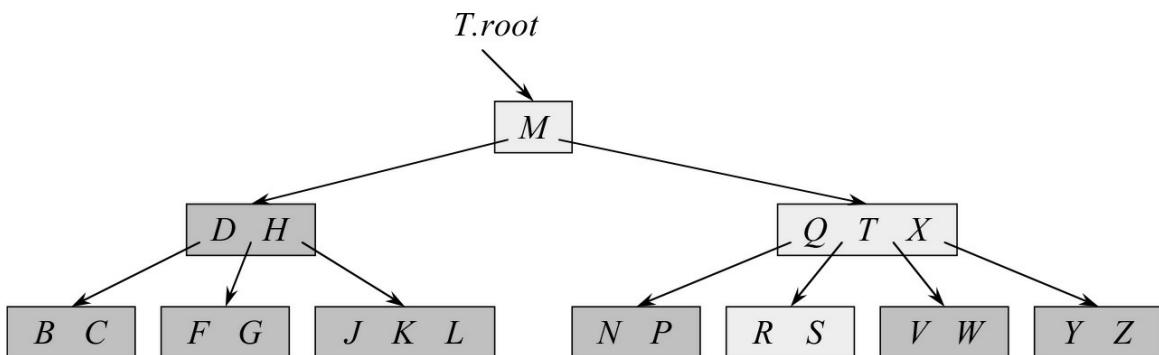
Observando que podemos superar o limite inferior $(n \lg n)$ para ordenação quando as chaves são inteiros dentro de uma faixa restrita, o Capítulo 20 questiona se podemos projetar uma estrutura de dados que suporta as operações de conjuntos dinâmicos Search, Insert, Delete, Minimum, Maximum, Successor e Predecessor no tempo $o(\lg n)$ quando as chaves são inteiros dentro de uma faixa restrita. Acontece que a resposta diz que podemos, se usarmos uma estrutura de dados recursiva conhecida como árvore de van Emde Boas. Se as chaves forem inteiros distintos extraídos do conjunto $\{0, 1, 2, \ldots, u - 1\}$, onde u é uma potência exata de 2, então as árvores de van Emde Boas suportam cada uma das operações citadas no tempo $O(\lg \lg u)$.

Finalmente, o Capítulo 21 apresenta estruturas de dados para conjuntos disjuntos. Temos um universo de n elementos que são particionados em conjuntos dinâmicos. Inicialmente, cada elemento pertence a seu próprio conjunto unitário. A operação Union une dois conjuntos, e a consulta Find-Set identifica o único conjunto que contém um determinado elemento no momento em questão. Representando cada conjunto por uma árvore enraizada simples, obtemos operações surpreendentemente rápidas: uma sequência de m operações é executada no tempo $O(m\ \alpha(n))$, onde $\alpha(n)$ é uma função de crescimento incrivelmente lento — $\alpha(n)$ é no máximo 4 em qualquer aplicação concebível. A análise amortizada que prova esse limite de tempo é tão complexa quanto a estrutura de dados é simples.

Os tópicos abordados nesta parte não são de modo algum os únicos exemplos de estruturas de dados "avançadas". Entre outras estruturas de dados avançadas citamos as seguintes:

- ***Árvores dinâmicas***, introduzidas por Sleator e Tarjan 319 e discutidas por Tarjan 330, mantêm uma floresta de árvores enraizadas disjuntas. Cada aresta em cada árvore tem um custo de valor real. Árvores dinâmicas suportam consultas para encontrar pais, raízes, custos de arestas e o custo de aresta mínimo em um caminho simples de um nó até uma raiz. As árvores podem ser manipuladas por corte de arestas, atualização de todos os custos de arestas em um caminho simples de um nó até uma raiz, ligação de uma raiz a uma outra árvore e transformação de um nó em raiz da árvore na qual ele aparece. Uma implementação de árvores dinâmicas dá um limite de tempo amortizado $O(\lg n)$ para cada operação; uma implementação mais complicada produz limites de tempo $O(\lg n)$ no pior caso. Árvores dinâmicas são usadas em alguns dos algoritmos de fluxo de rede assintoticamente mais rápidos.
- ***Árvores oblíquas***, desenvolvidas por Sleator e Tarjan 320 e, novamente, discutidas por Tarjan 330, são uma forma de árvore de busca binária, na qual as operações padrões de árvores de busca são executadas em tempo amortizado $O(\lg n)$. Uma das aplicações de árvores oblíquas simplifica árvores dinâmicas.
- Estruturas de dados ***persistentes*** permitem consultas, e, às vezes, também atualizações, em versões anteriores de uma estrutura de dados. Driscoll, Sarnak, Sleator e Tarjan 97 apresentam técnicas para transformar estruturas de dados ligadas em persistentes com apenas um pequeno custo de tempo e espaço. O Problema 13-1 dá um exemplo simples de um conjunto dinâmico persistente.
- Como veremos no Capítulo 20, várias estruturas de dados permitem uma implementação mais rápida de operações de dicionário (INSERT, DELETE e SEARCH) para um universo restrito de chaves. Tirando proveito dessas restrições, elas podem conseguir melhores tempos de execução assintóticos do pior caso que estruturas de dados baseadas em comparação. Fredman e Willard introduziram as ***árvores de fusão*** 115, as primeiras estruturas de dados a permitir operações de dicionário mais rápidas quando o universo está restrito a inteiros. Eles mostraram como implementar essas operações no tempo $O(\lg n/\lg \lg n)$. Várias estruturas de dados subsequentes, entre elas as ***árvores exponenciais de busca*** 16 também deram limites melhorados para algumas ou todas as operações de dicionário e são mencionadas em notas de capítulos em todo este livro.
- ***Estruturas de dados de grafos dinâmicos*** suportam várias consultas e ao mesmo tempo permitem que a estrutura de um grafo mude por meio de operações que inserem ou eliminam vértices ou arestas. Entre os exemplos das consultas que elas suportam citamos a conectividade de vértices 166, a conectividade de arestas, as árvores geradoras mínimas 165, a biconectividade e o fecho transitivo 164.

Notas do capítulo em todo o livro mencionam outras estruturas de dados adicionais.

---

[1] Como no Problema 10-2, definimos um heap intercalável para suportar Minimum e Extract-Min e, portanto, também podemos nos referir a ele como um ***heap de mínimo intercalável***. Alternativamente, se suportasse Maximum e Extract-Max, ele seria um ***heap de máximo intercalável***. A menos que especifiquemos o contrário, como padrão os heaps intercaláveis serão heaps de mínimo intercaláveis.

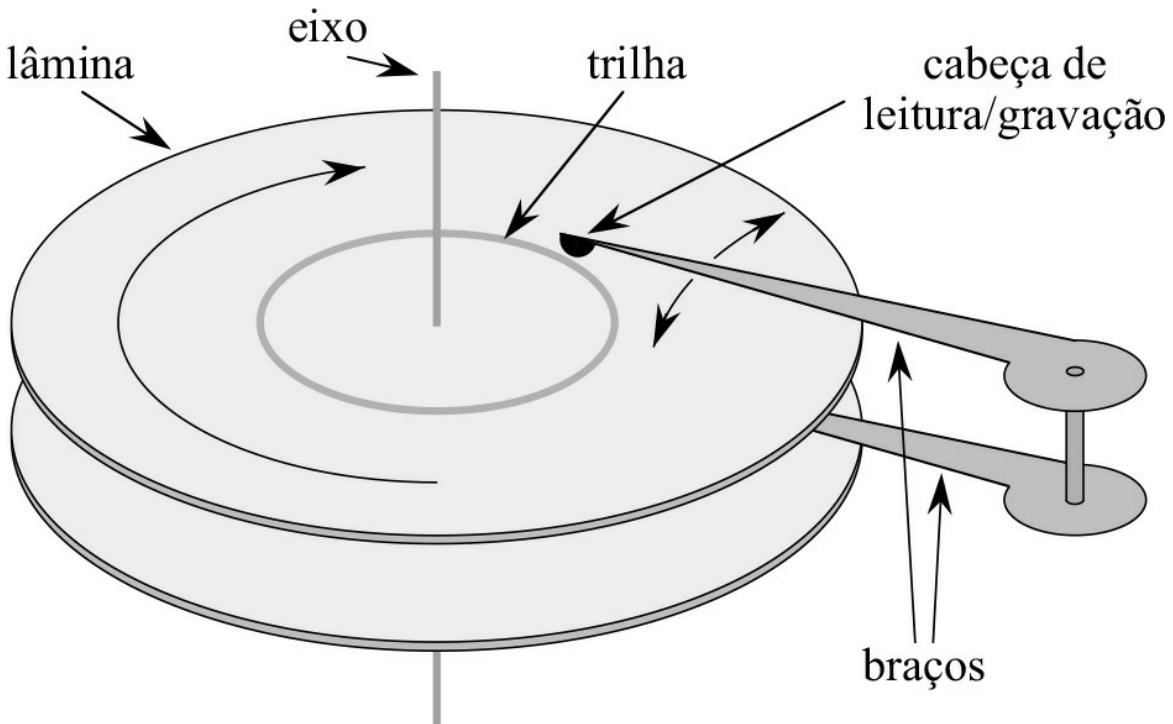
18 ÁRVORES B

B-árvores são árvores de busca balanceadas projetadas para funcionar bem em discos ou outros dispositivos de armazenamento secundário de acesso direto. B-árvores são semelhantes a árvores vermelho-preto (Capítulo 13), mas são melhores para minimizar operações de E/S de disco. Muitos sistemas de bancos de dados usam B-árvores ou variantes de B-árvores para armazenar informações.

As B-árvores são diferentes das árvores vermelho-preto no sentido de que os nós de B-árvores podem ter muitos filhos, de alguns até milhares. Isto é, o "fator de ramificação" de uma B-árvore pode ser bastante grande, embora normalmente, dependa de características da unidade de disco utilizada. As B-árvores são semelhantes às árvores vermelho-preto no sentido de que toda B-árvore de n nós tem altura $O(\lg n)$. Todavia, a altura exata de uma B-árvore pode ser consideravelmente menor que a altura de uma árvore vermelho-preto porque seu fator de ramificação e, por consequência, a base do logaritmo que expressa sua altura pode ser muito maior. Portanto, também podemos usar B-árvores para implementar muitas operações de conjuntos dinâmicos no tempo $O(\lg n)$.

B-árvores generalizam árvores de busca binária de modo natural. A Figura 18.1 mostra uma B-árvore simples. Se um nó interno x de uma B-árvore contém $x.n$ chaves, então x tem $x.n + 1$ filhos. As chaves no nó x servem como pontos de divisão que separam a faixa de chaves manipulada por x em $x.n + 1$ subfaixas, cada uma tratada por um filho de x. Quando procuramos uma chave em uma B-árvore, tomamos uma decisão de $(x.n + 1)$ vias, com base em comparações com as $x.n$ chaves armazenadas no nó x. A estrutura de nós de folhas é diferente da estrutura de nós internos; examinaremos essas diferenças na Seção 18.1.

A Seção 18.1 dá uma definição precisa de B-árvores e prova que o aumento da altura de uma B-árvore é apenas logarítmico de acordo com o número de nós que ela contém. A Seção 18.2 descreve como procurar uma chave e como inserir uma chave em uma B-árvore, e a Seção 18.3 discute eliminação. Porém, antes de prosseguir, precisamos perguntar por que avaliamos estruturas de dados projetadas para funcionar em um disco de modo diferente das estruturas de dados projetadas para funcionar em memória principal de acesso aleatório.

Figura 18.1 Uma B-árvore cujas chaves são as consoantes do alfabeto latino. Um nó interno x contendo $x.n$ chaves tem $x.n + 1$ filhos. Todas as folhas estão na mesma profundidade na árvore. Os nós sombreados em tom mais claro são examinados em uma busca pela letra R.

Estruturas de dados em armazenamento secundário

Os sistemas de computador aproveitam várias tecnologias que fornecem capacidade de memória. A ***memória primária*** (ou ***memória principal***) de um sistema de computador normalmente, consiste em chips de memória de silício. Em geral, essa tecnologia é mais de uma ordem de grandeza mais cara por bit armazenado que a tecnologia de armazenamento magnético, como fitas ou discos. A maioria dos sistemas de computador também tem ***armazenamento secundário*** baseado em discos magnéticos; muitas vezes, a quantidade de tal armazenamento secundário é no mínimo duas ordens de grandeza maior que a quantidade de memória primária. A Figura 18.2 mostra uma unidade de disco típica. A unidade consiste em uma ou várias ***lâminas*** que giram a uma velocidade constante em torno de um ***eixo*** comum. Um material magnetizável cobre a superfície de cada lâmina. A unidade de disco lê ou grava cada lâmina por meio de uma ***cabeça*** na extremidade de um ***braço***. Os braços podem movimentar suas cabeças aproximando-se ou afastando-se do fuso. Quando determinada cabeça está estacionária, a superfície que passa sob ela é denominada ***trilha***. Várias lâminas aumentam somente a capacidade da unidade de disco, mas não seu desempenho.

Embora os discos sejam mais baratos e tenham maior capacidade que a memória principal, eles são muito, muito mais lentos porque têm peças mecânicas móveis.[1] O movimento mecânico tem dois componentes: a rotação da lâmina e o movimento do braço. Na época da redação desta edição, a velocidade de rotação dos discos comerciais era 5.400-15.000 revoluções por minuto (RPM). O que existia, normalmente no comércio eram velocidades de 15.000 RPM em unidades de disco de grau de servidor, 7.200 RPM em unidades de disco de computadores de mesa e 5.400 RPM em unidades de disco de laptops. Embora a velocidade de 7.200 RPM pareça alta, uma rotação demora 8,33 milissegundos, tempo que é mais de cinco ordens de grandeza maior que os tempos de acesso de 50 nanossegundos (mais ou menos) comumente encontrados em memória de silício. Em outras palavras, se temos de esperar uma rotação completa para um item específico cair sob a cabeça de leitura/gravação, podemos acessar a memória principal mais de 100.000 vezes durante esse mesmo período. Em média, temos de esperar somente metade de uma rotação, mas ainda assim, a diferença entre tempos de acesso para memórias de silício e para discos é enorme. A movimentação dos braços também demora algum tempo. Na época em que redigimos esta edição, os tempos de acesso médios para discos comerciais estavam na faixa de 8-11 milissegundos.

Para amortizar o tempo gasto na espera de movimentos mecânicos, os discos acessam vários itens de cada vez, e não apenas um. As informações são divididas em várias ***páginas*** de bits de igual tamanho que aparecem consecutivamente dentro de trilhas, e cada leitura ou gravação de disco é de uma ou mais páginas inteiras. Para um disco típico, uma página pode ter 2^{11} a 2^{14} bytes de comprimento. Tão logo a cabeça de leitura/gravação esteja posicionada corretamente e o disco tenha girado até o início da página desejada, a leitura ou gravação em disco magnético é inteiramente eletrônica (exceto a rotação do disco) e o disco pode ler ou gravar rapidamente grande quantidade de dados.

Figura 18.2 Uma unidade de disco típica. Ela é composta por uma ou mais lâminas (aqui mostramos duas lâminas) que giram em torno de um fuso. Cada lâmina é lida e gravada com uma cabeça na extremidade de um braço. Os braços giram ao redor de um eixo pivô comum. Uma trilha é a superfície que passa sob a cabeça de leitura/gravação quando a cabeça está estacionária.

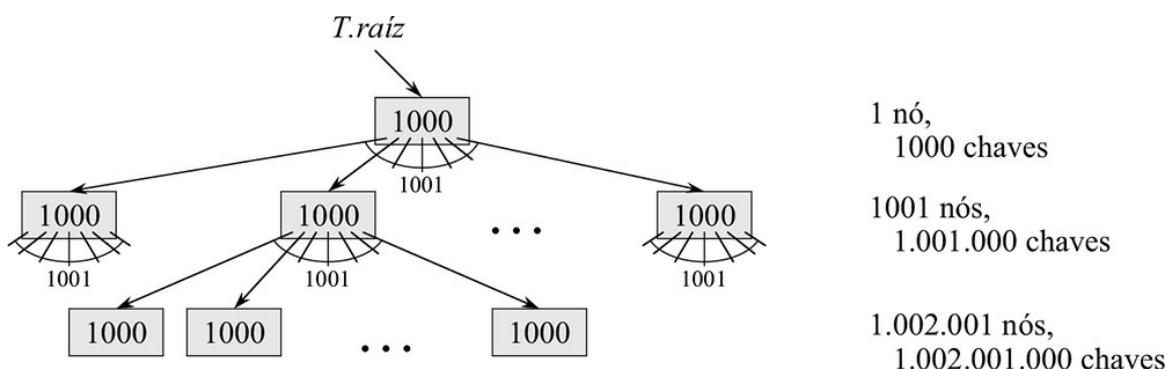
Muitas vezes, acessar uma página de informações em um disco e ler essa página demora mais que examinar todas as informações lidas. Por essa razão, neste capítulo estudaremos separadamente os dois componentes principais do tempo de execução:

- o número de acessos ao disco, e
- o tempo de CPU (ou de computação).

Medimos o número de acessos ao disco em termos do número de páginas de informações que precisam ser lidas do disco ou nele gravadas. Observamos que o tempo de acesso ao disco não é constante — depende da distância entre a trilha atual e a trilha desejada, e também da posição de rotação inicial do disco. Não obstante, usaremos o número de páginas lidas ou gravadas como uma aproximação de primeira ordem do tempo total gasto no acesso ao disco.

Em uma aplicação típica de B-árvore, a quantidade de dados manipulados é tão grande que os dados não cabem todos na memória principal de uma só vez. Os algoritmos de B-árvores copiam páginas selecionadas do disco para a memória principal conforme necessário e gravam novamente em disco as páginas que foram alteradas. Algoritmos de B-árvores mantêm somente um número constante de páginas na memória principal em qualquer instante; assim, o tamanho da memória principal não limita o tamanho das B-árvores que podem ser manipuladas.

Modelamos operações de disco em nosso pseudocódigo da maneira ilustrada a seguir. Seja x um ponteiro para um objeto. Se o objeto estiver atualmente na memória principal do computador, poderemos referenciar os atributos do objeto do modo usual: por exemplo, $x.chave$. Contudo, se o objeto referenciado por x residir no disco, teremos de executar a operação DISK-READ(x) para ler o objeto x para a memória principal antes de podermos referenciar seus atributos. (Supomos que, se x já está na memória principal, DISK-READ(x) não requer nenhum acesso ao disco; é uma "não operação" ("no-op").) De modo semelhante, a operação DISK-WRITE(x) é usada para gravar quaisquer alterações que tenham sido feitas nos atributos do objeto x. Isto é, o padrão típico de trabalho com um objeto é o seguinte:

x = um ponteiro para algum objeto
DISK-READ(x)
operações que acessam e/ou modificam os atributos de x
DISK-WRITE(x)   // omitida se nenhum atributo de x foi alterado
outras operações que acessam mas não modificam atributos de x

O sistema pode manter somente um número limitado de páginas na memória principal em qualquer instante. Suporemos que, o sistema descarrega da memória principal as páginas que não estão mais em uso; nossos algoritmos de B-árvores ignorarão essa questão.

Visto que na maioria dos sistemas o tempo de execução de um algoritmo de B-árvore depende primariamente do número de operações DISK-READ e DISK-WRITE que executa, seria bom que cada uma dessas operações lesse ou gravasse o máximo possível de informações. Assim, um nó de B-árvore é, normalmente, tão grande quanto uma página de disco inteira, e esse tamanho limita o número de filhos que um nó de B-árvore pode ter.

Para uma B-árvore grande armazenada em disco, normalmente, os fatores de ramificação estão entre 50 e 2.000, dependendo do tamanho de uma chave em relação ao tamanho de uma página. Um fator de ramificação grande reduz drasticamente a altura da árvore e também o número de acessos ao disco necessários para encontrar qualquer chave. A Figura 18.3 mostra uma B-árvore com um fator de ramificação de 1.001 e altura 2 que pode armazenar mais de um bilhão de chaves; não obstante, visto que podemos manter o nó de raiz permanentemente na memória principal, podemos encontrar qualquer chave dessa árvore com, no máximo, dois acessos ao disco.

Figura 18.3 Uma B-árvore de altura 2 contendo mais de um bilhão de chaves. Dentro de cada nó x aparece $x.n$, o número de chaves em x. Cada nó interno e folha contêm 1.000 chaves. Essa B-árvore contém 1.001 nós na profundidade 1 e mais de um milhão de folhas na profundidade 2.

18.1 DEFINIÇÃO DE B-ÁRVORES

Por questão de simplicidade, supomos — como fizemos para as árvores de busca binária e árvores vermelho-preto que quaisquer "informações satélites" associadas a uma chave residem no mesmo nó que a chave. Na prática, poderíamos até mesmo armazenar com cada chave apenas um ponteiro para uma outra página de disco que contenha as informações satélites para essa chave. O pseudocódigo neste capítulo, supõe implicitamente que as informações satélites associadas a uma chave, ou o ponteiro para tais informações satélites, acompanham a chave sempre que esta passar de nó para nó. Uma variante comum de B-árvore, conhecida como ***B-árvore***+, armazena todas as informações satélites nas folhas e apenas chaves e ponteiros de filhos nos nós internos, o que maximiza o fator de ramificação dos nós internos.

Uma ***B-árvore***+ T é uma árvore enraizada (cuja raiz é $T.raiz$) que tem as seguintes propriedades:

1. Todo nó x tem os seguintes atributos:

a. $x.n$, o número de chaves atualmente armazenadas no nó x,
   b. as próprias $x.n$ chaves, $x.chave_1$, $x.chave_2$, ... , $x.chave_{x.n}$, armazenadas em ordem não decrescente, de modo que $x.chave_1 \leq x.chave_2 \leq \ldots \leq x.chave_{x.n}$,
   c. $x.folha$, um valor booleano que é VERDADEIRO se x é uma folha e FALSO se x é um nó interno.

2. Cada nó interno x também contém $x.n + 1$ ponteiros $x.c_1, x.c_2, ..., x.c_{x.n+1}$ para seus filhos. Os nós de folhas não têm filhos e, assim, seus atributos c_i são indefinidos.
3. As chaves $x.chave_i$ separam as faixas de chaves armazenadas em cada subárvore: se k_i é qualquer chave armazenada na subárvore com raiz $c_i x$, então

$$k_1 \leq x.chave_1 \leq k_2 \leq x.chave_2 \leq ... \leq x.chave_{n[x]} \leq k_{x.n+1}\,.$$

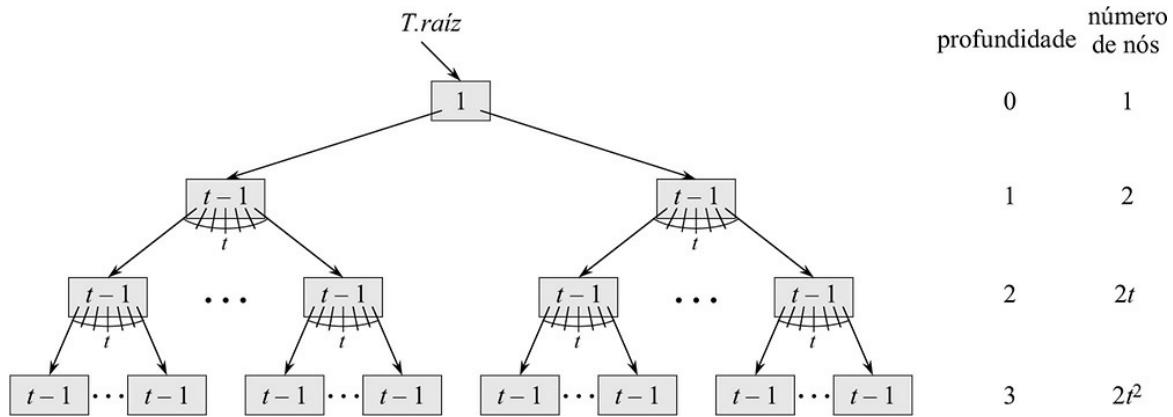
4. Todas as folhas têm a mesma profundidade, que é a altura h da árvore.
5. Os nós têm limites inferiores e superiores para o número de chaves que podem conter. Expressamos esses limites em termos de um inteiro fixo $t \geq 2$ denominado ***grau mínimo*** da B-árvore:
   a. Todo nó, exceto a raiz, deve ter no mínimo t - 1 chaves. Assim, todo nó interno, exceto a raiz, tem no mínimo t filhos. Se a árvore é não vazia, a raiz deve ter no mínimo uma chave.
   b. Todo nó pode conter no máximo $2t$ - 1 chaves. Portanto, um nó interno pode ter no máximo $2t$ filhos. Dizemos que um nó está ***cheio*** se contém exatamente $2t$ - 1 chaves.[2]

A B-árvore mais simples ocorre quando $t = 2$. Então, todo nó interno tem dois, três ou quatro filhos, e temos uma ***árvore 2-3-4***. Todavia, na prática, valores muito mais altos de t produzem B-árvores de menor altura.

A altura de uma B-árvore

O número de acessos ao disco exigidos para a maioria das operações em uma B-árvore é proporcional à altura da B-árvore. Analisamos agora a altura do pior caso de uma B-árvore.

Teorema 18.1

Se $n \geq 1$, então, para qualquer B-árvore T de n nós de altura h e grau mínimo $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2}.$$

Prova A raiz de uma B-árvore T contém no mínimo uma chave, e todos os outros nós contêm no mínimo t - 1 chaves. Assim, T, cuja altura é h, tem no mínimo dois nós na profundidade 1, no mínimo $2t$ nós na profundidade 2 e no mínimo $2t_2$ nós na profundidade 3 ,e assim por diante, até que na profundidade h ela tem no mínimo $2t_h$-1 nós. A Figura 18.4 ilustra tal árvore para $h = 3$. Assim, o número n de chaves satisfaz a desigualdade

$$\begin{aligned} n &\geq 1+(t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1+2(t-1)\left(\frac{t^h-1}{t-1}\right) \\ &= 2t^h-1 \end{aligned}$$

Por álgebra simples, obtemos $t_h \leq (n + 1)/2$. Tomando logaritmos em base t de ambos os lados prova-se o teorema.

Vemos aqui, o poder de B-árvores em comparação com árvores vermelho-preto. Embora a altura da árvore cresça na proporção $O(\lg n)$ em ambos os casos (lembre-se de que t é uma constante), para as B-árvores a base do logaritmo pode ser muitas vezes maior. Assim, B-árvores poupam um fator de aproximadamente $\lg t$ em relação a árvores vermelho-preto no que se refere ao número de nós examinados para a maioria das operações de árvore. Como, normalmente, temos de acessar o disco para examinar um nó arbitrário em uma árvore, as B-árvores evitam uma quantidade substancial de acessos ao disco.

Figura 18.4 Uma B-árvore de altura 3 contendo um número mínimo possível de chaves. Mostramos $x.n$ dentro de cada nó x.

Exercícios

18.1-1 Por que não permitimos um grau mínimo $t = 1$?

18.1-2 Para quais valores de t a árvore da Figura 18.1 é uma B-árvore válida?

18.1-3 Mostre todas as B-árvores válidas de grau mínimo 2 que representam $\{1, 2, 3, 4, 5\}$.

18.1-4 Qual é o número máximo de chaves que podem ser armazenadas em uma B-árvore de altura h em função do grau mínimo t?

18.1-5 Descreva a estrutura de dados que resultaria se cada nó preto em uma árvore vermelho-preto absorvesse seus filhos vermelhos incorporando os filhos vermelhos a seus próprios filhos pretos.

18.2 Operações básicas em B-árvores

Nesta seção, apresentamos os detalhes das operações B-Tree-Search, B-Tree-Create e B--Tree-Insert. Nesses procedimentos, adotamos duas convenções:

- A raiz da B-árvore está sempre na memória principal, de modo que nunca precisamos executar uma operação Disk-Read na raiz; porém, temos de executar uma operação Disk-Write da raiz, sempre que o nó de raiz for modificado.
- Qualquer nó só poderá ser passado como parâmetro após a execução de uma operação Disk-Read nesse mesmo nó.

Os procedimentos que apresentamos são algoritmos de "uma passagem" cuja execução ocorre na direção descendente em relação à raiz da árvore, sem ter de retornar.

Busca em uma B-árvore

Executar uma busca em uma B-árvore é muito semelhante a executar uma busca em uma árvore de busca binária, exceto que, em vez de tomar uma decisão de ramificação binária ou de “duas vias” em cada nó, tomamos uma decisão de ramificação de várias vias, de acordo com o número de filhos do nó. Mais exatamente, em cada nó interno x, tomamos uma decisão de ramificação de $(x.n + 1)$ vias.

B-TREE-SEARCH é uma generalização direta do procedimento TREE-SEARCH definido para árvores de busca binária. B-TREE-SEARCH toma como entrada um ponteiro para o nó de raiz x de uma subárvore e uma chave k que deve ser procurada nessa subárvore. Assim, a chamada de nível superior é da forma B-TREE-SEARCH($T.raiz$, k). Se k está na B-árvore, B-TREE-SEARCH retorna o par ordenado (y, i), que consiste em um nó y e um índice i tal que $y.chave_i = k$. Caso contrário, o procedimento retorna NIL.

```
B-TREE-SEARCH(x, k)
1   i = 1
2   while i ≤ x.n e k > x.chave_i
3       i = i + 1
4   if i ≤ x.n e k = x.chave_i
5       return (x, i)
6   elseif x.folha
7       return NIL
8   else DISK-READ(x.c_i)
9   return B-TREE-SEARCH(x.c_i, k)
```

Usando um procedimento de busca linear, as linhas 1-3 encontram o menor índice i tal que $k \leq x.chave_i$ ou definem i como $x.n + 1$. As linhas 4-5 verificam se agora descobrimos a chave e a retornam se a tivermos descoberto. Caso contrário, as linhas 6-9 terminam a busca sem sucesso (se x é uma folha) ou executam uma recursão para procurar a subárvore adequada de x, após executar a necessária operação DISK-READ naquele filho.

A Figura 18.1 ilustra a operação de B-TREE-SEARCH. O procedimento examina os nós sombreados em tom mais claro durante uma operação de busca da chave R.

Como ocorre no procedimento TREE-SEARCH para árvores de busca binária, os nós encontrados durante a recursão formam um caminho simples descendente desde a raiz da árvore. Portanto, o procedimento B-TREE-SEARCH acessa $O(h) = O(\log_t n)$ páginas de disco, onde h é a altura da B-árvore e n é o número de chaves na B-árvore. Visto que $x.n < 2t$, o laço **while** das linhas 2-3 demora o tempo $O(t)$ dentro de cada nó, e o tempo total de CPU é $O(th) = O(t \log_t n)$.

Criando uma B-árvore vazia

Para construir uma B-árvore T, primeiro utilizamos B-TREE-CREATE para criar um nó de raiz vazio e depois chamamos B-TREE-INSERT para acrescentar novas chaves. Esses dois procedimentos usam um procedimento auxiliar ALLOCATE-NODE, que aloca uma página de disco para ser usada como um novo nó no tempo $O(1)$. Podemos considerar que um nó criado por ALLOCATE-NODE não requer nenhuma operação DISK-READ, já que ainda não existe nenhuma informação útil armazenada no disco para esse nó.

```
B-TREE-CREATE(T)
1    x = ALLOCATE-NODE( )
2    x.folha = TRUE
3    x.n = 0
4    DISK-WRITE(x)
5    T.raiz = x
```

B-TREE-CREATE requer $O(1)$ operações de disco e tempo de CPU $O(1)$.

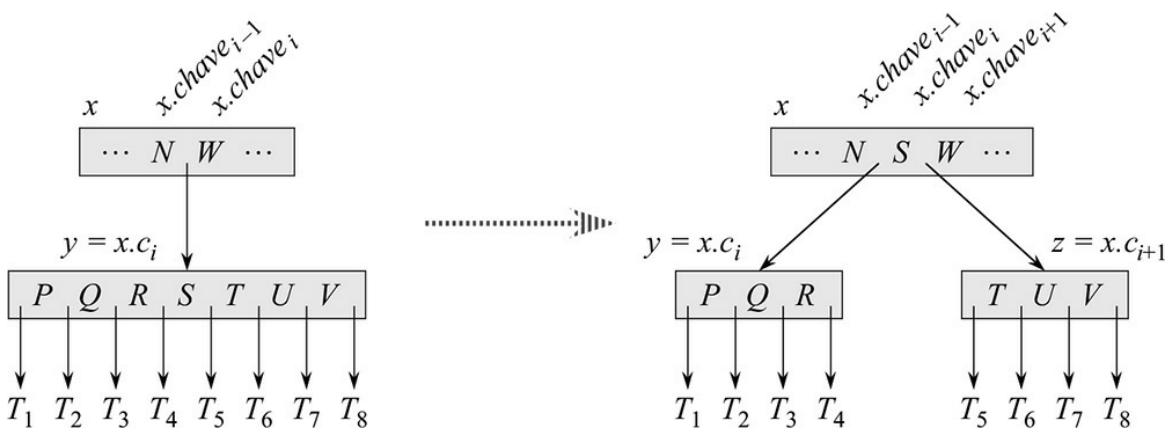
Inserindo uma chave em uma B-árvore

Inserir uma chave em uma B-árvore é significativamente mais complicado que inserir uma chave em uma árvore de busca binária. Quando se trata de árvores de busca binária, procuramos a posição de folha na qual inserir a nova chave. Porém, quando se trata de uma B-árvore, não podemos simplesmente criar um novo nó de folha e inseri-lo, já que a árvore resultante deixaria de ser uma B-árvore válida. Em vez disso, inserimos a nova chave em um nó de folha existente. Visto que não podemos inserir uma chave em um nó de folha que está cheio, recorremos a uma operação que ***reparte*** um nó cheio y (que tem $2t - 1$ chaves) em torno de sua ***chave mediana*** $chave_t[y]$ em dois nós que têm somente $t - 1$ chaves cada. A chave mediana sobe para dentro do pai de y para identificar o ponto de repartição entre as duas novas árvores. Porém, se o pai de y também está cheio, temos de reparti-lo antes de podermos inserir a nova chave e, assim, podemos acabar repartindo nós cheios por toda a árvore acima.

Como ocorre com uma árvore de busca binária, podemos inserir uma chave em uma B-árvore em uma única passagem descendente pela árvore da raiz até uma folha. Para tal, não esperamos para verificar se realmente precisamos repartir um nó cheio para executar a inserção. Em vez disso, à medida que descemos a árvore à procura da posição à qual pertence a nova chave, repartimos cada nó cheio que encontramos pelo caminho (inclusive a própria folha). Assim, sempre que queremos repartir um nó cheio y, temos a certeza de que seu pai não está cheio.

Repartindo um nó em uma B-árvore

O procedimento B-TREE-SPLIT-CHILD toma como entrada um nó interno x *não cheio* (que consideramos estar na memória principal) e um índice i tal que $x.c_i$ (que também consideramos estar na memória principal) que é um filho *cheio* de x. Então, o procedimento reparte esse filho em dois e ajusta x de modo que ele tenha um filho adicional. Para repartir uma raiz cheia, primeiro transformaremos a raiz em um filho de um novo nó de raiz vazio para podermos usar B-TREE-SPLIT-CHILD. Assim, a altura da árvore aumenta de uma unidade; repartir é o único meio de a árvore crescer.

A Figura 18.5 ilustra esse processo. Repartimos o nó cheio $y = x.c_i$ em torno de sua chave mediana S, que sobe para o nó x, pai de y. As chaves em y que são maiores que a chave mediana passam para um novo nó z, que se torna um novo filho de x.

```
B-TREE-SPLIT-CHILD(x, i)
1     z = ALLOCATE-NODE()
2     y = x.c_i
3     z.folha = y.folha
4     z.n = t − 1
5     for j = 1 to t − 1
6           z.chave_j = y.chave_{j+t}
7     if not y.folha
8           for j = 1 to t
9                 z.c_j = y.c_{j+t}
10    y.n = t − 1
11    for j = x.n + 1 downto i + 1
12          x.c_{j+1} − x.c_j
13    x.c_{i+1} = z
14    for j = x.n downto i
15          x.chave_{j+1} = x.chave_j
16    x.chave_i = y.chave_t
17    x.n = x.n + 1
18    DISK-WRITE(y)
19    DISK-WRITE(z)
20    DISK-WRITE(x)
```

B-TREE-SPLIT-CHILD funciona pelo método direto de “recortar e colar”. Aqui, x é o nó que está sendo repartido e y é o i-ésimo filho de x (definido na linha 2). O nó y tem originalmente $2t$ filhos ($2t$ - 1 chaves), mas é reduzido a t filhos (t - 1 chaves) por essa operação. O nó z toma os t maiores filhos (t - 1 chaves) de y, e z se torna um novo filho de x, posicionado logo após y na tabela de filhos de x. A chave mediana de y sobe e torna-se a chave em x que separa y e z.

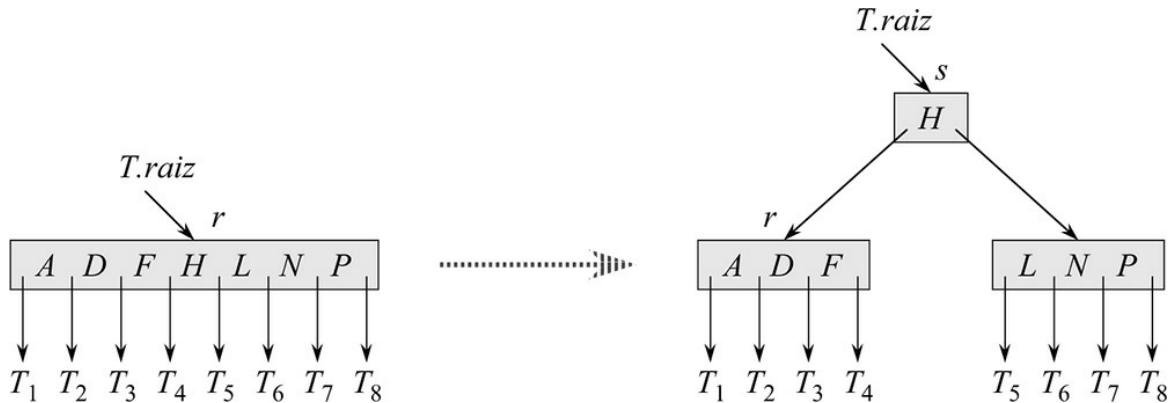
As linhas 1-9 criam o nó z e dão a ele as t - 1 chaves maiores e os t filhos correspondentes de y. A linha 10 ajusta a contagem de chaves para y. Finalmente, as linhas 11-17 inserem z como um filho de x, passam a chave mediana de y para cima até x, para separar y de z, e ajustam a contagem de chaves de x. As linhas 18-20 gravam todas as páginas de disco modificadas. O tempo de CPU usado por B-TREE-SPLIT-CHILD é (t), devido aos laços nas linhas 5-6 e 8-9. (Os outros laços são executados para $O(t)$ iterações.) O procedimento executa $O(1)$ operações de disco.

Figura 18.5 Repartição de um nó com $t = 4$. O nó $y = x.c_i$ é repartido em dois nós, y e z, e a chave mediana S de y sobe para dentro do pai de y.

Inserindo uma chave em uma B-árvore em uma única passagem descendente pela árvore

Inserir uma chave k em uma B-árvore T de altura h em uma única passagem descendente pela árvore requer $O(h)$ acessos ao disco. O tempo de CPU requerido é $O(th) = O(t \log_t n)$. O procedimento B-TREE-INSERT utiliza B-TREE-SPLIT-CHILD para garantir que a recursão nunca desça até um nó cheio.

```
B-TREE-INSERT(T,k)
1     r = T.raíz
2     if r.n == 2t − 1
3           s = ALLOCATE-NODE()
4           T.raíz = s
5           s.folha = FALSE
6           s.n = 0
7           s.c_1 = r
8           B-TREE-SPLIT-CHILD(s, 1)
9           B-TREE-INSERT-NONFULL(s, k)
10    else B-TREE-INSERT-NONFULL(r, k)
```

As linhas 3-9 tratam o caso no qual o nó de raiz r está cheio: a raiz é repartida e um novo nó s (que tem dois filhos) se torna a raiz. Repartir a raiz é o único modo de aumentar a altura de uma B-árvore. A Figura 18.6 ilustra esse caso. Diferentemente de uma árvore de busca binária, a altura de uma B-árvore aumenta em cima, em vez de embaixo. O procedimento termina chamando B-TREE-INSERT-NONFULL para inserir a chave k na árvore com raiz no nó de raiz não cheio. B-Tree-Insert-Nonfull executa recursão árvore abaixo conforme necessário e garante todas as vezes que o nó no qual executa a recursão não está cheio chamando B-TREE-SPLIT-CHILD quando necessário.

O procedimento recursivo auxiliar B-TREE-INSERT-NONFULL insere a chave k no nó x, suposto não cheio quando o procedimento é chamado. A operação de B-TREE-INSERT e a operação recursiva de B-TREE-INSERT-NONFULL garantem que tal suposição seja válida.

```
B-TREE-INSERT-NONFULL(x, k)
1     i = x.n
2     if x.folha
3           while i ≥ 1 e k < x.chave_i
4                 x.chave_{i+1} = x.chave_i
5                 i = i − 1
6           x.chave_{i+1} = k
7           x.n = x.n + 1
8           DISK-WRITE(x)
9     else while i ≥ 1 e k < x.chave_i
10                i = i − 1
11          i = i + 1
12          DISK-READ(x.c_i)
13          if x.c_i.n = 2t − 1
14                B-TREE-SPLIT-CHILD(x, i, x.c_i)
15                if k > x.chave_i
16                      i = i + 1
17          B-TREE-INSERT-NONFULL(x.c_i, k)
```

Figura 18.6 Repartição da raiz com $t = 4$. O nó de raiz r é repartido em dois, e um novo nó de raiz s é criado. A nova raiz contém a chave mediana de r e tem como filhos as duas metades de r. A altura da B-árvore aumenta de uma unidade quando a raiz é repartida.

O procedimento B-TREE-INSERT-NONFULL funciona da maneira descrita a seguir. As linhas 3-8 tratam o caso no qual x é um nó folha inserindo a chave k em x. Se x não é um nó folha, devemos inserir k no nó folha adequado na subárvore com raiz no nó interno x. Nesse caso, as linhas 9-11 determinam o filho de x para o qual a recursão é descendente. A linha 13 detecta se a recursão desceria até um filho cheio, caso em que a linha 14 usa B-TREE-SPLIT-CHILD para repartir esse filho em dois filhos não cheios, e as linhas 15-16 determinam qual dos dois filhos é agora o filho correto para o qual descer. (Observe que não há nenhuma necessidade de uma operação DISK-READ($x.c_i$) após a linha 16 incrementar i, já que nesse caso a recursão descerá até um filho que acabou de ser criado por B-TREE-SPLIT-CHILD.) Portanto, o efeito líquido das linhas 13-16 é garantir que o procedimento nunca executará uma recursão em um nó cheio. Então, a linha 17 executa recursão para inserir k na subárvore adequada. A Figura 18.7 ilustra os vários casos de inserção em uma B-árvore.

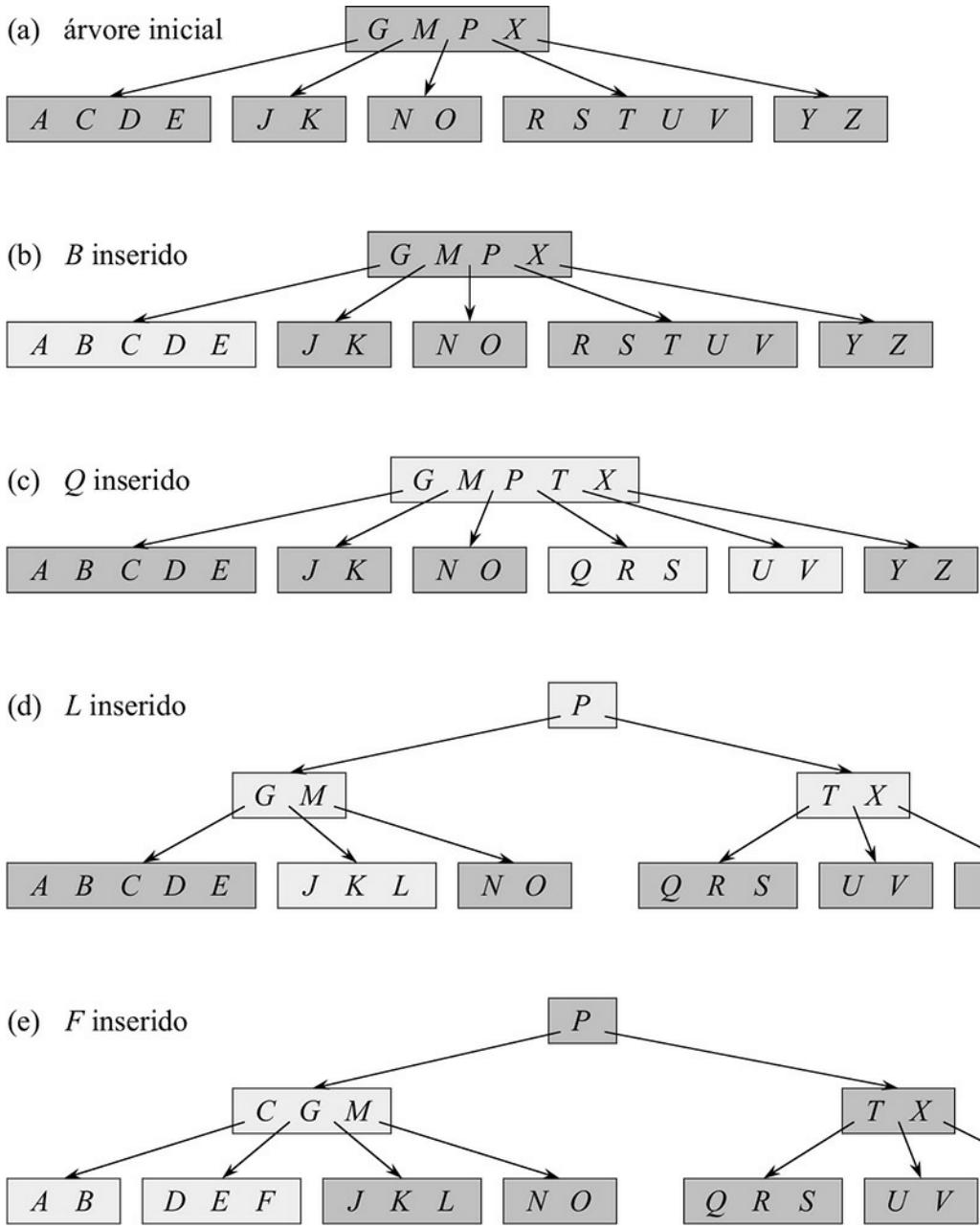
Para uma B-árvore de altura h, B-TREE-INSERT executa $O(h)$ acessos a disco, já que ocorrem somente $O(1)$ operações DISK-READ e DISK-WRITE entre chamadas a B-TREE-INSERT-NONFULL. O tempo total de CPU usado é $O(th) = O(t \log_t n)$. Visto que B-TREE-INSERT-NONFULL é recursivo de cauda, podemos implementá-lo alternativamente como um laço

while, demonstrando assim que o número de páginas que precisam estar na memória principal em qualquer instante é $O(1)$.

Exercícios

18.2-1 Mostre os resultados da inserção das chaves

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

em ordem, em uma B-árvore vazia com grau mínimo 2. Desenhe apenas as configurações da árvore imediatamente antes de algum nó ter de ser repartido, e desenhe também a configuração final.

18.2-2 Explique sob quais circunstâncias, se houver, ocorrem operações redundantes DISK-READ ou DISK-WRITE durante o curso da execução de uma chamada a B-TREE-INSERT. (Uma operação DISK-READ redundante é uma operação DISK-READ para uma página que já está na memória. Uma operação DISK-WRITE redundante grava em disco uma página de informações idêntica à que já está ali armazenada.)

18.2-3 Explique como encontrar a chave mínima armazenada em uma B-árvore e como encontrar o predecessor de uma determinada chave armazenada em uma B-árvore.

18.2-4 ★

Suponha que inserimos as chaves $\{1, 2, ..., n\}$ em uma B-árvore vazia com grau mínimo 2. Quantos nós tem a B-árvore final?

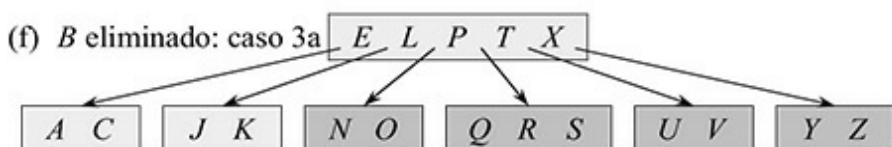
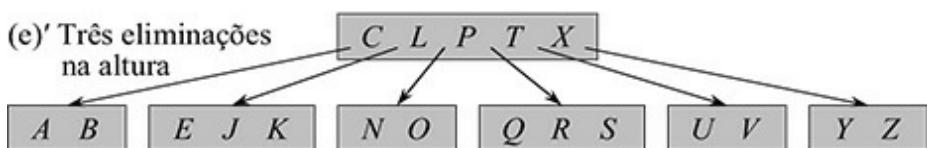
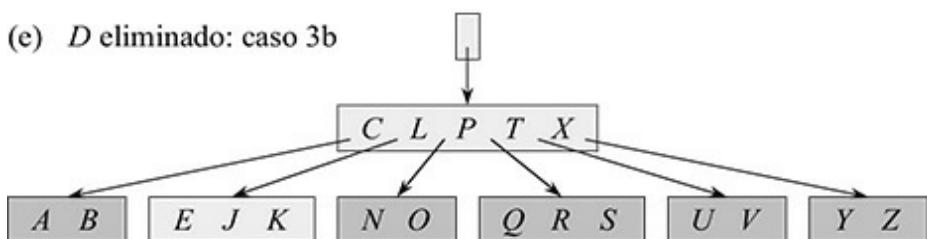
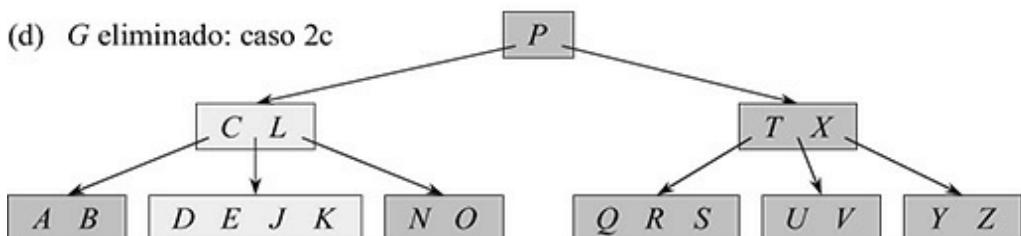
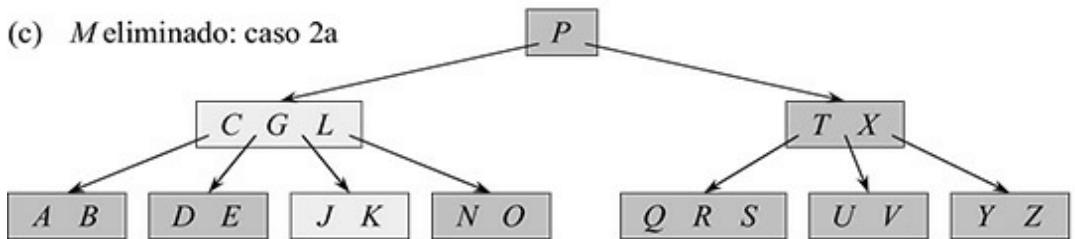
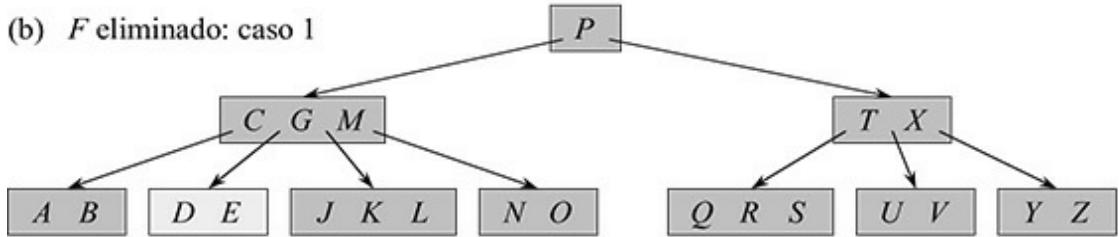
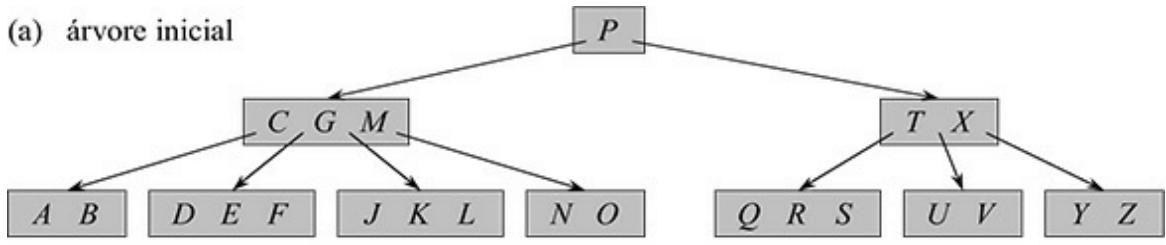
Figura 18.7 Inserção de chaves em uma B-árvore. O grau mínimo t para essa B-árvore é 3; assim, um nó pode conter no máximo cinco chaves. Nós que são modificados pelo processo de inserção estão sombreados em tom mais claro. **(a)** A árvore inicial para este exemplo. **(b)** O resultado da inserção de B na árvore inicial; essa é uma inserção simples em um nó de folha. **(c)** O resultado da inserção de Q na árvore anterior. O nó $R\ S\ T\ U\ V$ é repartido em dois nós contendo $R\ S$ e $U\ V$, a chave T é passada para cima até a raiz e Q é inserido na metade mais à esquerda das duas metades (o nó $R\ S$). **(d)** O resultado da inserção de L na árvore anterior. A raiz é repartida imediatamente, já que está cheia, e a altura da B-árvore aumenta de uma unidade. Então, L é inserido na folha que contém $J\ K$. **(e)** O resultado da inserção de F na árvore anterior. O nó $A\ B\ C\ D\ E$ é repartido antes de F ser inserido na metade mais à direita das duas metades (o nó $D\ E$).

18.2-5 Visto que nós folha não exigem nenhum ponteiro para filhos, eles poderiam usar um valor t diferente (maior) do número de nós internos para o mesmo tamanho de página de disco. Mostre como modificar os procedimentos de criação e inserção em uma B-árvore para tratar essa variação.

18.2-6 Suponha que quiséssemos implementar B-TREE-SEARCH para usar busca binária em vez de busca linear dentro de cada nó. Mostre que essa alteração resulta no tempo de CPU requerido $O(\lg n)$, independentemente do modo como t poderia ser escolhido em função de n.

18.2-7 Suponha que o hardware de disco nos permita escolher arbitrariamente o tamanho de uma página de disco, mas que o tempo necessário para ler a página de disco seja $a + bt$, onde a e b são constantes especificadas e t é o grau mínimo para uma B-árvore que utiliza páginas do tamanho selecionado. Descreva como escolher t para minimizar (aproximadamente) o tempo de busca da B-árvore. Sugira um valor ótimo de t para o caso em que a = 5 milissegundos e b = 10 microssegundos.

18.3 ELIMINAR UMA CHAVE EM UMA B-ÁRVORE

Eliminar uma chave em uma B-árvore procedimento análogo à inserção, mas é um pouco mais complicada porque podemos eliminar uma chave de qualquer nó — não apenas uma folha — e, quando eliminamos uma chave de um nó interno, temos de rearranjar os filhos do nó. Como na inserção, devemos nos prevenir para que a eliminação não produza uma árvore cuja estrutura viole as propriedades de B-árvores. Exatamente como tivemos de assegurar que um nó não ficasse demasiadamente grande devido à inserção, temos de garantir que um nó não fique demasiadamente pequeno durante a eliminação (exceto a raiz, que poder ter menos que o número mínimo t - 1 de chaves). Assim como um algoritmo de inserção simples poderia ter de retroceder se um nó no caminho até o ponto de inserção da chave estivesse cheio, uma abordagem de eliminação simples poderia ter de retroceder se um nó (exceto a raiz) no caminho até o ponto de eliminação da chave tivesse o número mínimo de chaves.

O procedimento B-TREE-DELETE elimina a chave k da subárvore com raiz em x. Projetamos esse procedimento para garantir que sempre que ele chamar a si mesmo recursivamente em um nó x o número de chaves em x é, pelo menos, o grau mínimo t. Observe que essa condição requer uma chave a mais que o mínimo exigido pelas condições usuais de B-árvores, de modo que, às vezes, será preciso passar uma chave para dentro de um nó filho, antes de a recursão descer até esse filho. Essa condição reforçada nos permite eliminar uma chave da árvore em uma única passagem descendente sem ter de "retroceder" (com uma única exceção, que explicaremos). Você deve interpretar a especificação dada a seguir para eliminação em uma B-árvore entendendo que, se alguma vez o nó de raiz x se tornar um nó interno sem nenhuma chave (essa situação pode ocorrer nos casos 2c e 3b, então eliminamos x, e o único filho de x, $x.c_1$, se torna a nova raiz da árvore, o que reduz a altura da árvore de uma unidade e preserva a propriedade de a raiz da árvore conter no mínimo uma chave (a menos que a árvore esteja vazia).

Descrevemos como a eliminação funciona, em vez de apresentarmos o pseudocódigo. A Figura 18.8 ilustra os vários casos de eliminação de chaves em uma B-árvore.

1. Se a chave k está no nó x e x é uma folha, elimine a chave k de x.
2. Se a chave k está no nó x e x é um nó interno, faça o seguinte:
    a. Se o filho y que precede k no nó x tem no mínimo t chaves, então encontre o predecessor k' de k na subárvore com raiz em y. Elimine recursivamente k', e substitua k por k' em x. (Podemos encontrar k' e eliminá-lo em uma única passagem descendente.)
    b. Se y tiver menos que t chaves, então, simetricamente, examine o filho z que segue k no nó x. Se z tiver no mínimo t chaves, encontre o sucessor k' de k na subárvore com raiz em z. Elimine k' recursivamente e substitua k por k' em x. (Podemos encontrar k' e eliminá-lo em uma única passagem descendente.)
    c. Caso contrário, y e z têm apenas t - 1 chaves, junte k e todo o z com y, de modo que x perde k e também o ponteiro para z, e agora y contém $2t$ - 1 chaves. Em seguida, libere z e elimine recursivamente k de y.

3. Se a chave k não estiver presente no nó interno x, determine a raiz $c_i x$ da subárvore adequada que deve conter k, se k estiver na árvore. Se $x.c_i$ tiver somente t - 1 chaves, execute a etapa 3a ou 3b conforme necessário para garantir que desceremos até um nó que contém no mínimo t chaves. Então, termine executando recursão no filho adequado de x.
    a. Se $x.c_i$ tiver somente t - 1 chaves, mas tiver um irmão imediato com no mínimo t chaves, dê a $x.c_i$ uma chave extra passando uma chave de x para baixo até $x.c_i$, passando uma chave do irmão imediato à esquerda ou à

direita de $x.c_i$ para cima até x, e passando o ponteiro de filho adequado do irmão para $x.c_i$.

b. Se $x.c_i$ e os irmãos imediatos de $x.c_i$ têm t - 1 chaves, junte $x.c_i$ com um irmão, o que envolve passar uma chave de x para baixo até o novo nó resultante da junção, que assim se torna a chave mediana para esse nó.

(a) árvore inicial
P
C G M
T X
A B
D E F
J K L
N O
Q R S
U V
Y Z
(b) F eliminado: caso 1
P
C G M
T X
A B
D E
J K L
N O
Q R S
U V
Y Z
(c) M eliminado: caso 2a
P
C G L
T X
A B
D E
J K
N O
Q R S
U V
Y Z
(d) G eliminado: caso 2c
P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z
(e) D eliminado: caso 3b
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
(e)′ Três eliminações na altura
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
(f) B eliminado: caso 3a
E L P T X
A C
J K
N O
Q R S
U V
Y Z

18.8 Eliminação de chaves em uma B-árvore. O grau mínimo para essa B-árvore é $t = 3$; portanto, um nó (exceto a raiz) não pode ter menos de duas chaves. Nós que são modificados estão sombreados em tom mais claro. (**a**) A B-árvore da Figura 18.7(e). (**b**) Eliminação de F. Esse é o caso 1: eliminação simples de uma folha. (**c**) Eliminação de M. Esse é o caso 2a: o predecessor L de M passa para cima e ocupa a posição de M. (**d**) Eliminação de G. Esse é o caso 2c: empurramos G para baixo para formar o nó $D\,E\,G\,J\,K$ e depois eliminamos G dessa folha (caso 1). (**e**) Eliminação de D. Esse é o caso 3b: a recursão não pode descer até o nó $C\,L$ porque ele tem apenas duas chaves; assim, empurramos P para baixo e o intercalamos com $C\,L$ e $T\,X$ para formar $C\,L\,P\,T\,X$; então, eliminamos D de uma folha (caso 1). (**e'**) Após (e), eliminamos a raiz, e a altura da árvore encolhe uma unidade. (**f**) Eliminação de B. Esse é o caso 3a: C é movido para preencher a posição de B e E é movido para preencher a posição de C.

Visto que a maioria das chaves em uma B-árvore se encontra nas folhas, podemos esperar que, na prática, operações de eliminação são usadas na maior parte das vezes para eliminar chaves de folhas. Então, o procedimento B-TREE-DELETE age em uma passagem descendente pela árvore, sem ter de retroceder. Contudo, quando elimina uma chave em um nó interno, o procedimento efetua uma passagem descendente pela árvore, mas pode ter de retornar ao nó do qual a chave foi eliminada para substituir a chave por seu predecessor ou sucessor (casos 2a e 2b).

Embora pareça complicado, esse procedimento envolve apenas $O(h)$ operações de disco para uma B-árvore de altura h, já que somente $O(1)$ chamadas a DISK-READ e DISK-WRITE são efetuadas entre invocações recursivas do procedimento. O tempo de CPU necessário é $O(th) = O(t \log_t n)$.

Exercícios

18.3-1 Mostre os resultados da eliminação de C, P e V, em ordem, da árvore da Figura 18.8(f).

18.3-2 Escreva o pseudocódigo para B-TREE-DELETE.

Problemas

18-1 Pilhas em armazenamento secundário

Considere a implementação de uma pilha em um computador que tem uma quantidade relativamente pequena de memória primária rápida e uma quantidade relativamente grande de armazenamento mais lento em disco. As operações PUSH e POP funcionam com valores de uma única palavra. A pilha que desejamos suportar pode crescer até seu tamanho tornar-se tão grande que não cabe mais na memória; por isso, parte dela tem de ser armazenada em disco.

Uma implementação de pilha simples, mas ineficiente, mantém a pilha inteira no disco. Mantemos na memória um ponteiro de pilha, que é o endereço de disco do elemento do topo da pilha. Se o ponteiro tiver o valor p, o elemento do topo é a (p mod m)-ésima palavra na página p/m do disco, onde m é o número de palavras por página. Para implementar a operação PUSH, incrementamos o ponteiro da pilha, lemos a página adequada para a memória de disco, copiamos o elemento a ser inserido na pilha para a palavra adequada na página e gravamos a página de novo no disco. Uma operação POP é semelhante. Decrementamos o ponteiro da pilha, lemos a página adequada no disco e voltamos ao topo da pilha. Não precisamos gravar de novo a página, já que ela não foi modificada.

Como operações de disco são relativamente caras, contamos dois custos para qualquer implementação: o número total de acessos ao disco e o tempo total de CPU. Qualquer acesso de disco a uma página de m palavras incorre em gastos de um acesso de disco e mais (m) tempo de CPU.

a. Assintoticamente, qual é o número de acessos ao disco do pior caso para n operações de pilhas usando essa implementação simples? Qual é o tempo de CPU para n operações de pilhas? (Expresse sua resposta em termos de m e n para esta parte e para as partes subsequentes.)

Agora, considere uma implementação de pilha na qual mantemos uma página da pilha na memória. (Mantemos também uma pequena quantidade de memória para controlar qual página está atualmente na memória.) Podemos executar uma operação de pilha somente se a página de disco relevante residir na memória. Se necessário, podemos gravar a página atualmente na memória no disco e ler a nova página do disco para a memória. Se a página de disco relevante já estiver na memória, não será necessário nenhum acesso ao disco.

b. Qual é o número de acessos ao disco do pior caso exigido para n operações PUSH? Qual é o tempo de CPU?

c. Qual é o número de acessos ao disco do pior caso exigido para n operações de pilha? Qual é o tempo de CPU?

Agora, suponha que implementemos a pilha mantendo duas páginas na memória (além de um pequeno número de palavras para contabilidade).

d. Descreva como gerenciar as páginas da pilha de modo que o número amortizado de acessos ao disco para qualquer operação de pilha seja $O(1/m)$ e o tempo de CPU amortizado para qualquer operação de pilha seja $O(1)$.

18-2 ***Junção e repartição de árvores 2-3-4***

A operação de ***junção*** toma dois conjuntos dinâmicos S' e S'' e um elemento x tal que, para qualquer $x' \in S'$ e $x'' \in S''$, temos $x'.chave < x.chave < x''.chave$. A junção retorna um conjunto $S = S' \cup \{x\} \cup S''$. A operação de ***repartição*** é como uma junção "inversa": dado um conjunto dinâmico S e um elemento $x \in S$, ela cria um conjunto S' que consiste em todos os elementos de $S - \{x\}$ cujas chaves são menores que $x.chave$, e um conjunto S'' que consiste em todos os elementos em $S - \{x\}$ cujas chaves são maiores que $x.chave$. Neste problema, investigaremos como implementar essas operações em árvores 2-3-4. Consideramos por conveniência que os elementos consistem apenas em chaves e que todos os valores de chaves são distintos.

a. Mostre como manter, para todo nó x de uma árvore 2-3-4, a altura da subárvore com raiz em x como um atributo $x.altura$. Certifique-se de que sua implementação não afeta os tempos de execução assintóticos de busca, inserção e eliminação.

b. Mostre como implementar a operação de junção. Dadas duas árvores 2-3-4 T' e T'' e uma chave k, a operação de junção deve ser executada no tempo $O(1 + |h' - h''|)$, onde h' e h'' são as alturas de T' e T'', respectivamente.

c. Considere o caminho simples p da raiz de uma árvore 2-3-4 T até uma dada chave k, o conjunto S' de chaves em T que são menores que k, e o conjunto S'' de chaves em T que são maiores que k. Mostre que p reparte S' em um conjunto de árvores $\{T'_0, T'_1, ..., k'_m\}$ e um conjunto de chaves $\{k'_1, k'_2, ..., k'_m\}$ onde, para $i = 1, 2, ..., m$, temos $y < k'_i < z$ para quaisquer chaves $y \in T'_{i-1}$ e $z \in T'_i$. Qual é a relação entre as alturas de T'_{i-1} e T'_i? Descreva o modo como p reparte S'' em conjuntos de árvores e chaves.

d. Mostre como implementar a operação de repartição em T. Utilize a operação de junção para montar as chaves de S' em uma única árvore 2-3-4 T' e as chaves de S'' em uma única árvore 2-3-4 T''. O tempo

de execução da operação de repartição deve ser $O(\lg n)$, onde n é o número de chaves em T. (*Sugestão*: Os custos para as operações de junção devem se cancelar.)

NOTAS DO CAPÍTULO

Knuth [211], Aho, Hopcroft e Ullman [5] e Sedgewick [306] apresentam discussões adicionais de esquemas de B-árvoresalanceadas e B-árvores. Comer [74] dá um levantamento abrangente de B-árvores. Guibas e Sedgewick [155] discutem as relações entre vários tipos de esquemas de B-árvoresalanceadas, inclusive árvores vermelho-preto e árvores 2-3-4.

Em 1970, J. E. Hopcroft criou as árvores 2-3, precursoras das B-árvores e das árvores 2-3-4, nas quais todo nó interno tem ou dois ou três filhos. Bayer e McCreight [35] apresentaram as B-árvores em 1972; eles não explicaram a escolha desse nome.

Bender, Demaine e Farach-Colton [40] estudaram como fazer B-árvores funcionarem bem na presença de efeitos de hierarquia de memória. Seus algoritmos ***sem consciência de cache*** funcionam eficientemente sem conhecer explicitamente os tamanhos de transferência de dados dentro da hierarquia de memória.

---

[1] Na época da redação desta edição, unidades de disco de estado sólido tinham acabado de chegar ao mercado de consumo. Embora sejam mais rápidas que as unidades de disco mecânicas, custam mais por gigabyte e têm menor capacidade que as unidades de disco mecânicas.

[2] Uma outra variante comum de uma B-árvore, conhecida como ***B-árvore****, exige que cada nó interno esteja no mínimo 2/3 cheio, em vez de no mínimo metade cheio, como exige uma B-árvore.

19 HEAPS DE FIBONACCI

A estrutura de dados heap de Fibonacci tem dupla finalidade. A primeira é suportar um conjunto de operações que constitui o que é conhecido como "heap intercalável". Em segundo lugar, várias operações de heap de Fibonacci são executadas em tempo amortizado constante, o que torna essa estrutura bem adequada para aplicações que invocam tais operações frequentemente.

Heaps intercaláveis

Um ***heap intercalável*** é qualquer estrutura de dados que suporte as cinco operações seguintes, nas quais cada elemento tem uma *chave*:

MAKE-HEAP( ) cria e retorna um novo heap que não contém nenhum elemento.

INSERT(H, x) insere o elemento x, cuja *chave* já foi preenchida, no heap H.

MINIMUM(H) retorna um ponteiro para o elemento do heap H cuja chave é mínima.

EXTRACT-MIN(H) elimina o elemento do heap H cuja chave é mínima, retornando um ponteiro para o elemento.

UNION(H_1, H_2) cria e retorna um novo heap que contém todos os elementos dos heaps H_1 e H_2. Os heaps H_1 e H_2 são "destruídos" por essa operação.

Além das operações de heap intercalável citadas, os heaps de Fibonacci também suportam as duas operações seguintes:

DECREASE-KEY(H, x, k) atribui ao elemento x dentro do heap H o novo valor de chave k, que supomos não ser maior que seu valor de chave atual.[1]

DELETE(H, x) elimina o elemento x do heap H.

Como mostra a tabela na Figura 19.1, se não precisamos da operação UNION, heaps binários comuns, como os que são utilizados em heapsort (Capítulo 6), funcionam razoavelmente bem. Outras operações, exceto a operação UNION, são executadas no tempo do pior caso $O(\lg n)$ em um heap binário. Contudo, se precisarmos dar suporte à operação UNION, o desempenho dos heaps binários é sofrível. Como concatena os dois arranjos que contêm os heaps binários que serão intercalados e depois executa BUILD-MIN-HEAP (veja Seção 6.3), a operação UNION demora o tempo Q(n) no pior caso.

Heaps de Fibonacci, por outro lado, têm limites de tempo assintótico melhores que os heaps binários para as operações INSERT, UNION e DECREASE-KEY, e os mesmos tempos de execução assintóticos para as operações restantes. Observe, entretanto, que os tempos de execução de heaps de Fibonacci na Figura 19.1 são limites de tempo

amortizado e não limites de tempo do pior caso por operação. A operação UNION demora somente tempo amortizado constante em um heap de Fibonacci, o que é significativamente melhor que o tempo linear do pior caso exigido em um heap binário (considerando, é claro, que um limite de tempo amortizado seja suficiente).

Heaps de Fibonacci na teoria e na prática

De um ponto de vista teórico, os heaps de Fibonacci são especialmente desejáveis quando o número de operações EXTRACT-MIN e DELETE é pequeno em relação ao número de outras operações executadas. Essa situação surge em muitas aplicações. Por exemplo, alguns algoritmos para problemas de grafos podem chamar DECREASE-KEY uma vez por aresta. Para grafos densos, que têm muitas arestas, o tempo amortizado Q(1) de cada chamada de DECREASE-KEY significa uma grande melhoria em relação ao tempo do pior caso Q(lg n) de heaps binários. Algoritmos rápidos para problemas como cálculo de árvores geradoras mínimas (Capítulo 23) e localização de caminhos mais curtos de origem única (Capítulo 24) tornam essencial o uso de heaps de Fibonacci.

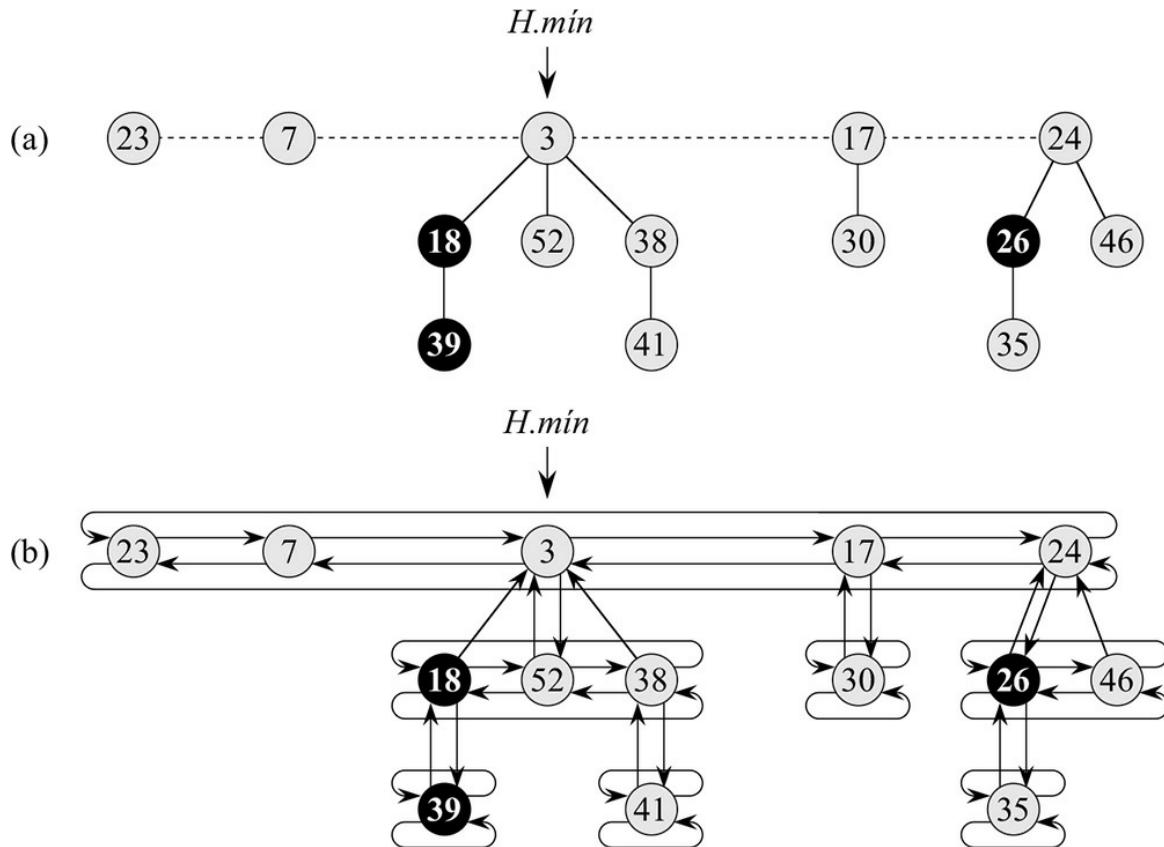
Porém, do ponto de vista prático, os fatores constantes e a complexidade da programação de heaps de Fibonacci os tornam menos desejáveis que heaps binários (ou k-ários) comuns para a maioria das aplicações. Assim, os heaps de Fibonacci são predominantemente de interesse teórico. Se uma estrutura de dados muito mais simples com os mesmos limites de tempo amortizado que os heaps de Fibonacci fosse desenvolvida, ela também seria de utilidade prática.

Heaps binários e heaps de Fibonacci são ineficientes no suporte da operação SEARCH; pode demorar um pouco para encontrar um elemento com determinada chave. Por essa razão, operações como DECREASE-KEY e DELETE, que se referem a um dado elemento, requerem um ponteiro para esse elemento como parte de sua entrada. Como em nossa discussão de filas de prioridades na Seção 6.5, quando usamos um heap intercalável em uma aplicação, muitas vezes, armazenamos um descritor para o objeto da aplicação correspondente em cada elemento de heap intercalável, bem como um descritor para o elemento de heap intercalável correspondente em cada objeto da aplicação. A natureza exata desses descritores depende da aplicação e de sua implementação.

Assim, como várias outras estruturas de dados que vimos, heaps de Fibonacci são baseados em árvores enraizadas. Representamos cada elemento por um nó dentro de uma árvore, e cada nó tem um atributo *chave*. No restante deste capítulo, usaremos o termo "nó" em vez de "elemento". Ignoraremos também questões de alocação de nós antes de inserção e liberação de nós após eliminação; em vez disso, suporemos que o código que chama os procedimentos de heap cuida desses detalhes.

A Seção 19.1 define heaps de Fibonacci, discute sua representação e apresenta a função potencial usada para sua análise amortizada. A Seção 19.2 mostra como implementar as operações de heaps intercaláveis e como obter os limites de tempo amortizado mostrados na Figura 19.1. As duas operações restantes, DECREASE-KEY e DELETE, são apresentados na Seção 19.3. Finalmente, a Seção 19.4 conclui uma parte fundamental da análise e também explica o curioso nome da estrutura de dados.

Procedimento	Heap binário (pior caso)	Heap de Fibonacci (amortizado)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Figura 19.1 Tempos de execução para operações em duas implementações de heaps intercaláveis. O número de itens no(s) heap(s) no momento de uma operação é denotado por n.

19.1 Estrutura de heaps de Fibonacci

Um ***heap de Fibonacci*** é uma coleção de árvores enraizadas que estão ***ordenadas por heap de mínimo***. Isto é, cada árvore obedece à ***propriedade de heap de mínimo***: a chave de um nó é maior ou igual à chave de seu pai. A Figura 19.2(b) mostra um exemplo de heap de Fibonacci.

Como mostra a Figura 19.2(b), cada nó x contém um ponteiro $x.p$ para seu pai e um ponteiro $x.filho$ para algum de seus filhos. Os filhos de x estão interligados em uma lista circular, duplamente ligada, que denominamos ***lista de filhos*** de x. Cada filho y em uma lista de filhos tem ponteiros $y.esquerda$ e $y.direita$ que apontam para os irmãos à esquerda e à direita de y, respectivamente. Se o nó y é um filho único, então $y.esquerda = y.direita = y$. Filhos podem aparecer em qualquer ordem em uma lista de filhos.

Listas circulares, duplamente ligadas (veja Seção 10.2) têm duas vantagens para utilização em heaps de Fibonacci. A primeira é que podemos inserir um nó em qualquer ponto ou remover um nó de qualquer lugar de uma lista circular duplamente ligada no tempo $O(1)$. A segunda é que, dadas duas dessas listas, podemos concatená-las (ou "entrelaçá-las") em uma única lista circular duplamente ligada no tempo $O(1)$. Nas descrições de operações de heaps de Fibonacci, faremos referência a essas operações informalmente, deixando a cargo do leitor preencher os detalhes de suas implementações como quiser.

Cada nó tem dois outros atributos. Armazenamos o número de filhos na lista de filhos do nó x em $x.grau$. O atributo do valor booleano $x.marca$ indica se o nó x perdeu um filho desde a última vez que x se tornou o filho de um outro nó. Os nós recém-criados não estão marcados, e um nó x se torna desmarcado sempre que passa a ser o filho de outro nó. Até examinarmos a operação Decrease-Key na Seção 19.3, simplesmente definiremos todos os atributos *marca* como False.

Figura 19.2 (**a**) Um heap de Fibonacci que consiste em cinco árvores ordenadas por heaps de mínimo e 14 nós. A linha tracejada indica a lista de raízes. O nó mínimo do heap é o nó que contém a chave 3. Nós pretos são nós marcados. O potencial desse heap de Fibonacci particular é $5 + 2 \times 3 = 11$. (**b**) Uma representação mais completa que mostra ponteiros p (setas para cima), *filho* (setas para baixo) e

esquerda e *direita* (setas laterais). Esses detalhes são omitidos nas figuras restantes deste capítulo, já que todas as informações mostradas aqui podem ser determinadas pelo que aparece na parte (a).

Acessamos determinado heap de Fibonacci *H* por um ponteiro *H.min* para a raiz de uma árvore que contém uma chave mínima; esse nó é denominado ***nó mínimo*** do heap de Fibonacci. Se mais de uma raiz tiver uma chave como o valor mínimo, qualquer raiz pode servir como o nó mínimo. Quando um heap de Fibonacci *H* está vazio, *H.min* = NIL.

As raízes de todas as árvores em um heap de Fibonacci são interligadas por meio de seus ponteiros *esquerda* e *direita* em uma lista circular duplamente ligada denominada ***lista de raízes*** do heap de Fibonacci. Assim, o ponteiro *H.min* aponta para o nó na lista de raízes cuja chave é mínima. As árvores podem aparecer em qualquer ordem dentro de uma lista de raízes. Contamos com um outro atributo para um heap de Fibonacci *H*: *H.n*, o número de nós atualmente em H.

Função potencial

Como mencionamos, usaremos o método do potencial da Seção 17.3 para analisar o desempenho de operações de heap de Fibonacci. Para dado heap de Fibonacci *H*, indicamos por $t(H)$ o número de árvores na lista de raízes de *H* e por $m(H)$ o número de nós marcados em *H*. Então, definimos o potencial (H) do heap de Fibonacci *H* por

$$\Phi(H) = t(H) + 2m(H) . \tag{19.1}$$

(Entenderemos melhor essa função potencial na Seção 19.3.) Por exemplo, o potencial do heap de Fibonacci mostrado na Figura 19.1 é $5 + 2 \cdot 3 = 11$. O potencial de um conjunto de heaps de Fibonacci é a soma dos potenciais dos heaps de Fibonacci que o constituem. Suporemos que uma unidade de potencial pode pagar uma quantidade constante de trabalho, onde a constante é suficientemente grande para cobrir o custo de qualquer das peças de trabalho específicas de tempo constante que poderíamos encontrar.

Consideramos que uma aplicação de um heap de Fibonacci começa sem nenhum heap. Então, o potencial inicial é 0 e, pela equação (19.1), o potencial é não negativo em todas as vezes subsequentes. Pela equação (17.3), um limite superior para o custo total amortizado é um limite superior para o custo total real para a sequência de operações.

Grau máximo

As análises amortizadas que realizaremos nas seções restantes deste capítulo supõem que conhecemos um limite superior $D(n)$ para o grau máximo de qualquer nó em um heap de Fibonacci de *n* nós. Não provaremos isso, mas quando somente as operações de heaps intercaláveis são suportadas, $D(n) \le \lg n$. (O Problema 19-2(d) pede que você prove essa propriedade.) Nas Seções 19.3 e 19.4, mostraremos que, quando suportamos também DECREASE-KEY e DELETE, $D(n) = O(\lg n)$.

19.2 OPERAÇÕES DE HEAPS INTERCALÁVEIS

As operações de heaps intercaláveis em heaps de Fibonacci retardam o trabalho o máximo possível. As várias operações têm permutas de desempenho. Por exemplo, inserimos um nó acrescentando-o à lista de raízes, o que demora apenas tempo constante. Se começássemos com um heap de Fibonacci vazio e então inseríssemos *k* nós, o heap de Fibonacci consistiria em apenas uma lista de raízes de *k* nós. Em troca, se executarmos em seguida uma operação EXTRACT-MIN no heap de Fibonacci *H*, após remover o nó para o qual *H.min* aponta, teremos de examinar cada um dos k - 1 nós restantes na lista de raízes para encontrar o novo nó mínimo. Ao mesmo tempo que temos de percorrer toda a lista de raízes durante a operação EXTRACT-MIN, também consolidamos nós em árvores ordenadas por heaps de mínimo para reduzir o tamanho da lista de raízes. Veremos que, não importando como era a lista antes de uma

operação EXTRACT-MIN, depois dela cada nó na lista de raízes tem um grau que é exclusivo dentro da lista de raízes, o que resulta em uma lista de tamanho no máximo $D(n) + 1$.

Criando um novo heap de Fibonacci

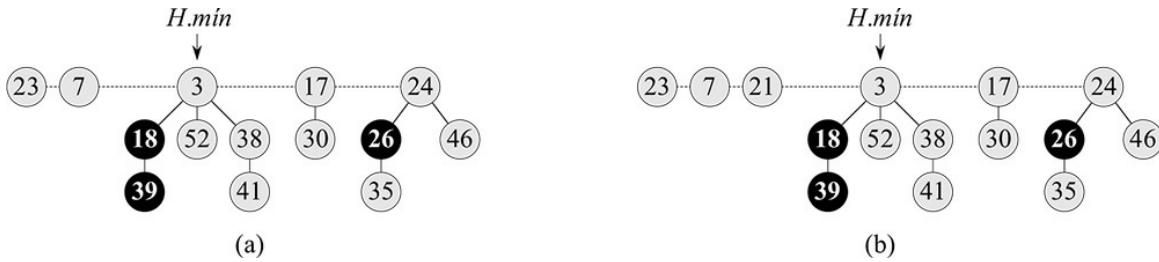
Para tornar um heap de Fibonacci vazio, o procedimento MAKE-FIB-HEAP aloca e devolve o objeto heap de Fibonacci H, onde $H.n = 0$ e $H.min$ = NIL; não há nenhuma árvore em H. Como $t(H) = 0$ e $m(H) = 0$, o potencial do heap de Fibonacci vazio é $F(H) = 0$. Assim, o custo amortizado de MAKE-FIB-HEAP é igual ao seu custo real $O(1)$.

Inserindo um nó

O procedimento a seguir insere o nó x no heap de Fibonacci H, supondo que o nó x já tenha sido alocado e que $x.chave$ já tenha sido preenchida.

```
FIB-HEAP-INSERT(H, x)
1     x.grau = 0
2     x.p = NIL
3     x.filho = NIL
4     x.marca = FALSE
5     if H.min == NIL
6           crie a lista de raízes de H contendo só x
7           H.min = x
8     else insira x na lista de raízes de H
9           if x.chave < H.min.chave
10                H.min = x
11    H.n = H.n + 1
```

As linhas 1-4 inicializam alguns dos atributos estruturais do nó x. A linha 5 testa para verificar se o heap de Fibonacci H está vazio. Se estiver, as linhas 6-7 fazem de x o único nó na lista de H e definem $H.min$ para apontar para x. Caso contrário, as linhas 8-10 inserem x na lista de raízes de H e atualizam $H.min$ se necessário. Finalmente, a linha 11 incrementa *H.n para refletir a adição de um novo nó. A* Figura 19.3 mostra um nó com chave 21 inserida no heap de Fibonacci da Figura 19.2.

Para determinar o custo amortizado de FIB-HEAP-INSERT, seja H o heap de Fibonacci de entrada e H' o heap de Fibonacci resultante. Então, $t(H') = t(H) + 1$ e $m(H') = m(H)$, e o aumento em potencial é

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1 .$$

Como o custo real é $O(1)$, o custo amortizado é $O(1) + 1 = O(1)$.

Encontrando o nó mínimo

O nó mínimo de um heap de Fibonacci H é dado pelo ponteiro $H.min$; assim, podemos encontrar o nó mínimo no tempo real $O(1)$. Visto que o potencial de H não muda, o custo amortizado dessa operação é igual ao seu custo real $O(1)$.

Figura 19.3 Inserindo um nó em um heap de Fibonacci. (**a**) Um heap de Fibonacci H. (**b**) O heap de Fibonacci H após a inserção do nó com chave 21. O nó se torna sua própria árvore ordenada por heap de mínimo e é então acrescentado à lista de raízes, tornando-se o irmão à esquerda da raiz.

Unindo dois heaps de Fibonacci

O procedimento a seguir une os heaps de Fibonacci H_1 e H_2, destruindo H_1 e H_2 no processo. Ele simplesmente concatena as listas de raízes de H_1 e H_2, e determina o novo nó mínimo. Depois disso, os objetos que representam H_1 e H_2 nunca mais serão usados.

```
FIB-HEAP-UNION(H1, H2)
1    H = MAKE-FIB-HEAP( )
2    H.min = H1.min
3    concatenar a lista de raízes de H2 com a lista de raízes de H
4    if (H1.min = NIL) ou (H2.min ≠ NIL e H2.min < H1.min)
5         H.min = H1.min
6    H.n = H1.n + H2.n
7    return H
```

As linhas 1-3 concatenam as listas de raízes de H_1 e H_2 em uma nova lista de raízes H. As linhas 2, 4 e 5 calculam o nó mínimo de H, e a linha 6 calcula $H.n$ como o número total de nós. A linha 7 devolve o heap de Fibonacci resultante H. Como no procedimento FIB-HEAP-INSERT, todas as raízes permanecem raízes.

A mudança de potencial é

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) \\ &= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\ &= 0\ ,\end{aligned}$$

porque $t(H) = t(H_1) + t(H_2)$ e $m(H_1) + m(H_2)$. Assim, o custo amortizado de FIB-HEAP-UNION é igual ao seu custo real $O(1)$.

Extraindo o nó mínimo

O processo de extrair o nó mínimo é a mais complicada das operações apresentadas nesta seção. É também aqui, que o trabalho adiado de consolidar árvores na lista de raízes finalmente ocorre. O pseudocódigo a seguir extrai o nó mínimo. O código supõe por conveniência que, quando um nó é removido de uma lista ligada, os ponteiros restantes na lista são atualizados, mas os ponteiros no nó extraído se mantêm inalterados. O código também chama o procedimento auxiliar CONSOLIDATE, que veremos em breve.

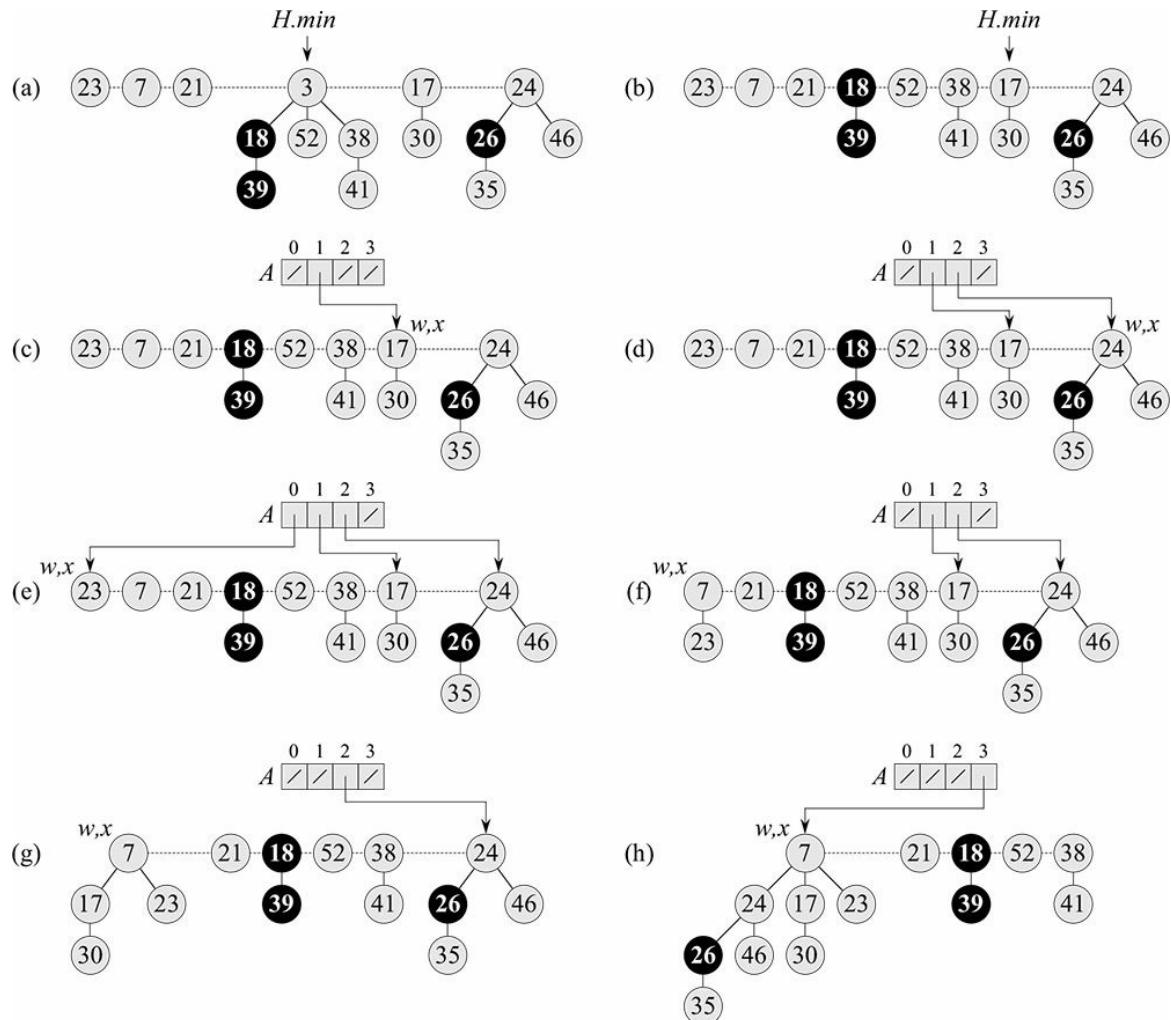
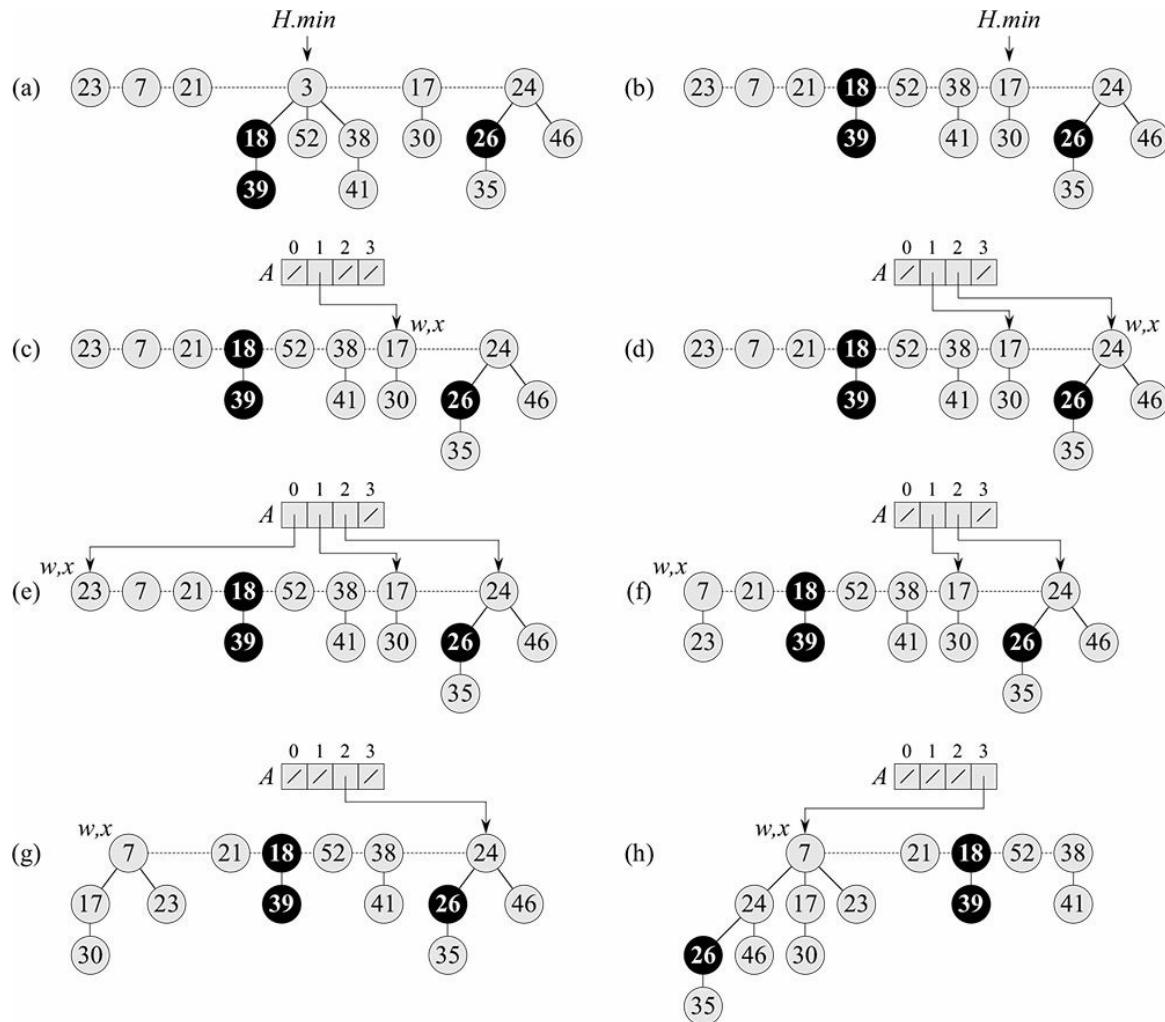
```
FIB-HEAP-EXTRACT-MIN(H)
1   z = H.min
2   if z ≠ NIL
3       for cada filho x de z
4           adicionar x à lista de raízes de H
5           x.p = NIL
6       remova z da lista de raízes de H
7       if z = z.direita
8           H.min = NIL
9       else H.min = z.direita
10          CONSOLIDATE(H)
11      H.n = H.n − 1
12  return z
```

Como mostra a Figura 19.4, FIB-HEAP-EXTRACT-MIN funciona primeiro criando uma raiz a partir de cada um dos filhos do nó mínimo e removendo o nó mínimo da lista de raízes. Então, consolida a lista de raízes, ligando raízes de mesmo grau até restar no máximo uma raiz de cada grau.

Começamos na linha 1 gravando um ponteiro z para o nó mínimo; o procedimento devolve esse ponteiro no final. Se z = NIL, então o heap de Fibonacci H já está vazio, e terminamos. Caso contrário, eliminamos o nó z de H, transformando todos os filhos de z em raízes de H nas linhas 3-5 (inserindo-os na lista de raízes) e removendo z da lista de raízes na linha 6. Se z for seu próprio irmão à direita depois da linha 6, então z era o único nó na lista de raízes e não tinha nenhum filho; assim, resta tornar o heap de Fibonacci vazio na linha 8 antes de retornar z. Caso contrário, fazemos $H.min$ apontar para uma outra raiz diferente de z (nesse caso, o filho *à direita de z*), que não será necessariamente o novo nó mínimo quando FIB-HEAP-EXTRACT-MIN termina. A Figura 19.4(b) mostra o heap de Fibonacci da Figura 19.4(a) após a execução da linha 9.

A próxima etapa, na qual reduzimos o número de árvores no heap de Fibonacci, é ***consolidar*** a lista de raízes de H, o que é realizado pela chamada a CONSOLIDATE(H). Consolidar a lista de raízes consiste em executar repetidamente as etapas seguintes até que toda raiz na lista de raízes tenha um valor de *grau* distinto.

1. Encontre duas raízes x e y na lista de raízes com o mesmo grau. Sem prejuízo da generalidade, seja $x.chave \leq y.chave$.
2. ***Ligue*** y a x: remova y da lista de raízes e torne y um filho de x chamando o procedimento FIB-HEAP-LINK. Esse procedimento incrementa o atributo $x.grau$ e limpa a marca em y.

O procedimento CONSOLIDATE usa um arranjo auxiliar $A[0 .. D(H.n)]$ para controlar as raízes de acordo com seus graus. Se $A[i] = y$, então y é atualmente uma raiz com $y.grau = i$. É claro que, para alocar o arranjo temos de saber como calcular o limite superior $D(H.n)$ no grau máximo, mas veremos como fazer isso na Seção 19.4.

```
CONSOLIDATE(H)
1   seja A[0 . .D(H.n)] um novo arranjo
2   for i = 0 to D(H.n)
3       A[i] = NIL
4   for cada nó w na lista de raízes de H
5       x = w
6       d = x.grau
7       while A[d] ≠ NIL
8           y = A[d]              // um outro nó com o mesmo grau de x
9           if x.chave > y.chave
10              trocar x com y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i ] ≠ NIL
18          if H.min == NIL
19              criar uma nova lista de raízes para H contendo apenas A[i]
20              H.min = A[i]
21          else inserir A[i] na lista de raízes de H
22              if A[i].chave < H.min.chave
23                  H.min = A[i]
```

```
FIB-HEAP-LINK(H, y, x)
1   remover y da lista de raízes de H
2   tornar y um filho de x, incrementando x.grau
3   y.marca = FALSE
```

Em detalhes, o procedimento CONSOLIDATE funciona como descrevemos a seguir. As linhas 1-3 alocam e inicializam o arranjo A fazendo cada entrada NIL. O laço **for** das linhas 4-14 processa cada raiz w na lista de raízes. À medida que interligamos raízes, w pode ser ligada a algum outro nó e deixar de ser uma raiz. Não obstante, w está sempre em uma árvore com raiz em algum nó x, que pode ser ou não a própria w. Como queremos, no máximo uma raiz para cada grau, examinamos o arranjo A para verificar se ele contém uma raiz y com o mesmo grau de x. Se contiver, ligamos as raízes x e y, mas garantindo que x permaneça uma raiz após a ligação. Isto é, ligamos y a x depois de permutar os ponteiros para as duas raízes se a chave de y for menor do que a chave de x. Após ligarmos y a x, o grau de x aumentou de 1, e assim continuamos esse processo, ligando x e uma outra raiz cujo grau seja igual ao novo grau de x, até que nenhuma outra raiz que tivermos processado tenha o mesmo grau de x. Então, fazemos a entrada adequada de A apontar para x, de modo que, ao processarmos raízes mais adiante, teremos gravado que x é a única raiz de seu grau que já processamos. Quando esse laço **for** termina, restará no máximo uma raiz de cada grau, e o arranjo A apontará para cada raiz restante.

Figura 19.4 A ação de FIB-HEAP-EXTRACT-MIN. (**a**) Um heap de Fibonacci *H*. (**b**) A situação após a remoção do nó mínimo *z* é da lista de raízes e a adição de seus filhos à lista de raízes. (**c**)-(**e**) O arranjo *A* e as árvores após cada uma das três primeiras iterações do laço **for** das linhas 4-14 do procedimento CONSOLIDATE. O procedimento processa a lista de raízes começando no nó apontado por *H.min* e seguindo os ponteiros *à direita*. Cada parte mostra os valores de *w* e *x* no fim de uma iteração. (**f**)-(**h**) A próxima iteração do laço **for**, com os valores de *w* e *x* mostrados no fim de cada iteração do laço **while** das linhas 7-13. A parte (f) mostra a situação após a primeira passagem pelo laço **while**. O nó com chave 23 foi ligado ao nó com chave 7, que agora é apontado por x. Na parte (g), o nó com chave 17 foi ligado ao nó com chave 7, o qual ainda é apontado por *x*. Na parte (h), o nó com chave 24 foi ligado ao nó com chave 7. Como nenhum nó foi apontado anteriormente por *A*3, no fim da iteração do laço **for**, *A*3 é feito apontar para a raiz da árvore resultante.

Figura 19.4, **continuação** (**i**)-(**l**) A situação após cada uma das quatro iterações seguintes do laço **for**. (**m**) O heap de Fibonacci *H* após a reconstrução da lista de raízes pelo arranjo *A* e pela determinação do novo ponteiro *minH*.

O laço **while** das linhas 7-13 liga repetidamente a raiz *x* da árvore que contém o nó *w* a uma outra árvore cuja raiz tem o mesmo grau que *x*, até que nenhuma outra raiz tenha o mesmo grau. Esse laço **while** mantém o seguinte invariante:

No início de cada iteração do laço **while**, $d = x.grau$.

Usamos esse invariante de laço assim:

Inicialização: A linha 6 assegura que o invariante de laço é válido na primeira vez em que entramos no laço.

Manutenção: Em cada iteração do laço **while**, $A[d]$ aponta para alguma raiz *y*. Como $d = x.grau = y.grau$, queremos ligar *x* e *y*. Entre *x* e *y*, a que tiver a menor chave se tornará o pai da outra, como resultado da operação de ligação e, assim, as linhas 9-10 trocam os ponteiros para *x* e *y*, se necessário. Em seguida, ligamos *x* a *y* pela chamada FIB-HEAP-LINK(H, y, x) na linha 11. Essa chamada incrementa $x.grau$, mas deixa $y.grau$ como *d*. O nó *y* não é mais uma raiz e, portanto, a linha 12 remove o ponteiro para ele no arranjo *A*. Como a chamada de FIB-HEAP-LINK incrementa o valor de $x.grau$, a linha 13 restaura o invariante $d = x.grau$.

Término: Repetimos o laço **while** até $A[d] =$ NIL; nesse caso não existe nenhuma outra raiz com o mesmo grau que *x*.

Depois que o laço **while** termina, definimos $A[d]$ como *x* na linha 14 e executamos a próxima iteração do laço **for**.

As Figuras 19.4(c)-(e) mostram o arranjo *A* e as árvores resultantes após as três primeiras iterações do laço **for** das linhas 4-14. Na próxima iteração do laço **for**, ocorrem três ligações; seus resultados são mostrados nas Figuras

19.4(f)-(h). As Figuras 19.4(i)-(l) mostram o resultado das quatro iterações seguintes do laço **for**.

Agora, só falta a limpeza. Uma vez terminado o laço **for** das linhas 4-14, a linha 15 esvazia a lista de raízes, e as linhas 16-23 recriam a lista pelo arranjo A. O heap de Fibonacci resultante é mostrado na Figura 19.4(m). Após consolidar a lista de raízes, FIB-HEAP-EXTRACT-MIN termina decrementando $H.n$ na linha 11 e retornando um ponteiro para o nó eliminado z na linha 12.

Agora, estamos prontos para mostrar que o custo amortizado de extrair o nó mínimo de um heap de Fibonacci de n nós é $O(D(n))$. Seja H a representação do heap de Fibonacci imediatamente antes da operação FIB-HEAP-EXTRACT-MIN.

Começamos com o custo real de extrair o nó mínimo. Uma contribuição $O(D(n))$ vem de FIB-HEAP-EXTRACT-MIN processar no máximo $D(n)$ filhos do nó mínimo e do trabalho nas linhas 2-3 e 16-23 de CONSOLIDATE. Resta analisar a contribuição do laço **for** das linhas 4-14. O tamanho da lista de raízes na chamada a CONSOLIDATE é no máximo $D(n) + t(H) - 1$, já que ela consiste nos $t(H)$ nós originais da lista de raízes, menos o nó de raiz extraído, mais os filhos do nó extraído, que são no máximo $D(n)$. Dentro de uma dada iteração do laço **for** das linhas 4-14, o número de iterações do laço **while** das linhas 7-13 depende da lista de raízes. Porém, sabemos que toda vez que passamos pelo laço **while** uma das raízes é ligada a uma outra raiz; assim, o número total de iterações do laço **while** para todas as iterações do laço **for** é, no máximo, proporcional a $D(n) + t(H)$. Portanto, o trabalho total real na extração do nó mínimo é $O(D(n) + t(H))$.

O potencial antes da extração do nó mínimo é $t(H) + 2m(H)$, e o potencial depois disso é no máximo $(D(n) + 1) + 2m(H)$, já que restam no máximo $D(n) + 1$ raízes, e nenhum nó foi marcado durante a operação. Assim, o custo amortizado é, no máximo,

$$\begin{aligned} & O(D(n) + t(H)) + ((D(n) + 1\ ) + 2m(H)) - (t(H) + 2m(H)) \\ & \quad = O(D(n)) + O(t(H)) - t(H) \\ & \quad = O(D(n))\ , \end{aligned}$$

visto que podemos ajustar a escala das unidades de potencial para dominar a constante oculta em $O(t(H))$. Intuitivamente, o custo da execução de cada ligação é pago pela redução do potencial, já que a ligação reduz o número de raízes em uma unidade. Veremos na Seção 19.4 que $D(n) = O(\lg n)$, de modo que o custo amortizado de extrair o nó mínimo é $O(\lg n)$.

Exercícios

19.2-1 Mostre o heap de Fibonacci resultante de chamar FIB-HEAP-EXTRACT-MIN para o heap de Fibonacci mostrado na Figura 19.4(m).

19.3 DECREMENTAR UMA CHAVE E ELIMINAR UM NÓ

Nesta seção, mostraremos como decrementar a chave de um nó em um heap de Fibonacci no tempo amortizado $O(1)$ e como eliminar qualquer nó de um heap de Fibonacci de n nós no tempo amortizado $O(D(n))$. Na Seção 19.4, mostraremos que o grau máximo $D(n)$ é $O(\lg n)$, o que implicará que FIB-HEAP-EXTRACT-MIN e FIB-HEAP-DELETE são executados no tempo amortizado $O(\lg n)$.

Decrementando uma chave

No pseudocódigo a seguir para a operação FIB-HEAP-DECREASE-KEY, supomos, como antes, que remover um nó de uma lista ligada não muda quaisquer dos atributos estruturais no nó removido.

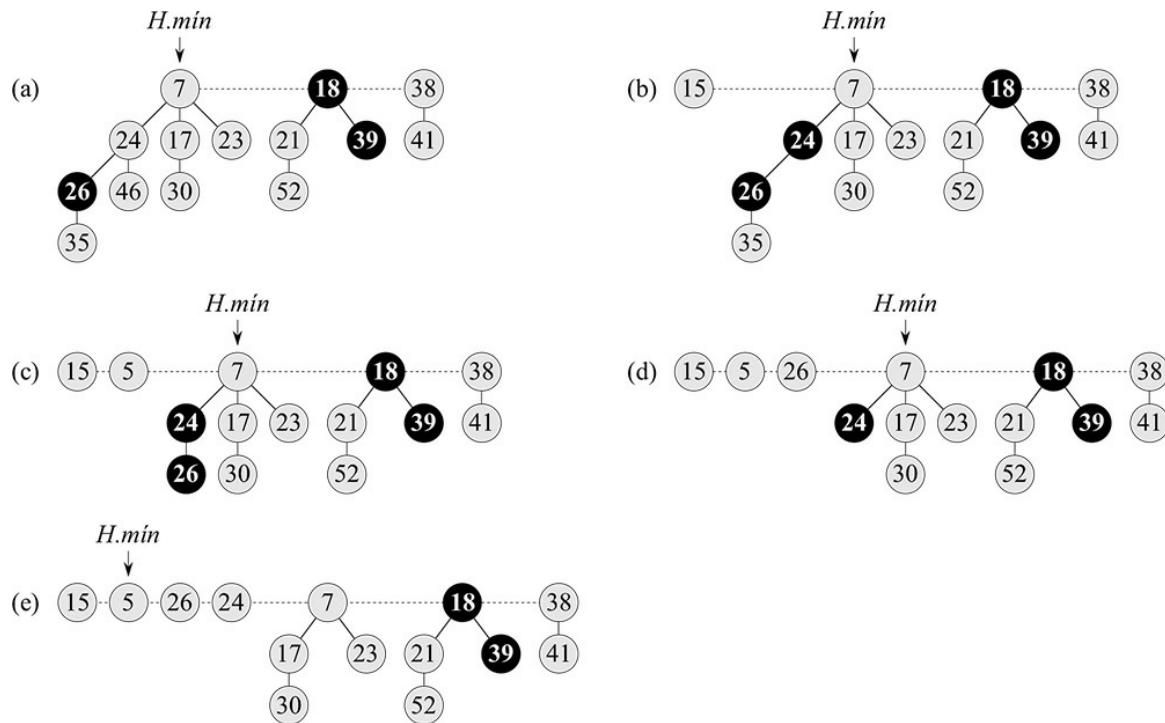
```
FIB-HEAP-DECREASE-KEY(H, x, k)
1   if k > x.chave
2       error "nova chave é maior que chave atual"
3   x.chave = k
4   y = x.p
5   if y ≠ NIL e x.chave < y.chave
6       CUT(H, x, y)
7       CASCADING-CUT(H, y)
8   if0 x.chave < H.min.chave
9       H.min = x

CUT(H, x, y)
1   remover x da lista de filhos de y decrementando y.grau
2   adicionar x à lista de raízes de H
3   x.p = NIL
4   x.marca = FALSE

CASCADING-CUT(H, y)
1   z = y.p
2   if z ≠ NIL
3       if y.marca == FALSE
4           y.marca = TRUE
5       else CUT(H, y, z)
6           CASCADING-CUT(H, z)
```

O procedimento FIB-HEAP-DECREASE-KEY funciona da maneira descrita a seguir. As linhas 1-3 asseguram que a nova chave não é maior que a chave atual de x, e designam a nova chave a x. Se x é uma raiz ou se $x.chave \geq y.chave$, onde y é pai de x, então não precisa ocorrer nenhuma mudança estrutural, já que a ordenação por heap de mínimo não foi violada. As linhas 4-5 testam essa condição.

Se a ordenação por heap de mínimo foi violada, muitas mudanças podem ocorrer. Começamos ***cortando*** x na linha 6. O procedimento CUT "corta" a ligação entre x e seu pai y, fazendo de x uma raiz.

Usamos os atributos *marca* para obter os limites de tempo desejados. Eles registram uma pequena fração da história de cada nó. Suponha que os eventos a seguir tenham ocorrido com o nó x:

1. em algum momento, x era uma raiz,
2. depois x foi ligado (foi feito filho de) a outro nó,
3. então dois filhos de x foram removidos por cortes.

Tão logo o segundo filho tenha sido perdido, cortamos x de seu pai, fazendo dele uma nova raiz. O atributo $x.marca$ é TRUE se as etapas 1 e 2 ocorreram e um filho de x foi cortado. Então, o procedimento CUT limpa $x.marca$ na linha 4, já que ele executa a etapa 1. (Agora, podemos ver por que a linha 3 de FIB-HEAP-LINK limpa $y.marca$: o nó y está sendo ligado a um outro nó e, assim, a etapa 2 está sendo executada. Na próxima vez que um filho de y for cortado, $y.marca$ será definido como TRUE.)

Ainda não terminamos, porque x poderia ser o segundo filho cortado de seu pai y, desde o tempo em que y esteve ligado a outro nó. Portanto, a linha 7 de FIB-HEAP-DECREASE-KEY executa uma operação de ***corte em cascata*** em y. Se y é uma raiz, então o teste na linha 2 de CASCADING-CUT faz o procedimento simplesmente retornar. Se y for não marcado, o procedimento o marca na linha 4, já que seu primeiro filho acabou de ser cortado, e retorna. Contudo, se y for marcado, ele acabou de perder seu segundo filho; y é cortado na linha 5, e CASCADING-CUT chama a si mesmo recursivamente na linha 6 para o pai de y, z. O procedimento CASCADING-CUT sobe a árvore recursivamente até encontrar uma raiz ou um nó não marcado.

Tão logo tenham ocorrido todos os cortes em cascata, as linhas 8-9 de FIB-HEAP-DECREASE- KEY terminam atualizando $H.min$ se necessário. O único nó cuja chave mudou foi o nó x, já que sua chave decresceu. Assim, o novo nó mínimo é o nó original ou é o nó x.

A Figura 19.5 mostra a execução de duas chamadas a FIB-HEAP-DECREASE-KEY, começando com o heap de Fibonacci mostrado na Figura 19.5(a). A primeira chamada, mostrada na Figura 19.5(b), não envolve nenhum corte em cascata. A segunda chamada, mostrada nas Figuras 19.5(c)-(e), invoca dois cortes em cascata.

Agora, mostraremos que o custo amortizado de FIB-HEAP-DECREASE-KEY é apenas $O(1)$. Começamos determinando seu custo real. O procedimento FIB-HEAP-DECREASE-KEY demora o tempo $O(1)$, mais o tempo para executar os cortes em cascata. Suponha que uma determinada invocação de FIB-HEAP-DECREASE-KEY resulte em c chamadas de CASCADING-CUT (a chamada feita da linha 7 de FIB-HEAP-DECREASE-KEY seguida por c - 1 chamadas recursivas de CASCADING-CUT). Cada chamada de CASCADING-CUT demora o tempo $O(1)$ sem incluir as chamadas recursivas. Assim, o custo real de FIB-HEAP-DECREASE-KEY, incluindo todas as chamadas recursivas, é $O(c)$.

Em seguida, calculamos a mudança no potencial. Denotamos por H o heap de Fibonacci imediatamente antes da operação de FIB-HEAP-DECREASE-KEY. A chamada a CUT na linha 6 de FIB-HEAP-DECREASE-KEY cria uma nova árvore enraizada no nó x e limpa o bit de marcação de x (que pode já ter sido FALSE). Cada chamada recursiva de CASCADING-CUT, com exceção da última, corta um nó marcado e limpa o bit de marcação. Depois disso, o heap de Fibonacci contém $t(H) + c$ árvores (as $t(H)$ árvores originais, $c - 1$ árvores produzidas por cortes em cascata e a árvore com raiz em x) e no máximo $m(H) - c + 2$ nós marcados ($c - 1$ foram desmarcados por cortes em cascata, e a última chamada de CASCADING-CUT pode ter marcado um nó). Então, a mudança no potencial é no máximo

Figura 19.5 Duas chamadas de FIB-HEAP-DECREASE-KEY. (**a**) O heap de Fibonacci inicial. (**b**) O nó com chave 46 tem sua chave decrementada para 15. O nó se torna uma raiz, e seu pai (com chave 24), que antes não estava marcado, se torna marcado. (**c**)-(**e**) O nó com chave 35 tem sua chave decrementada para 5. Na parte (c), o nó, agora com chave 5, se torna uma raiz. Seu pai, com chave 26, está marcado, e assim ocorre um corte em cascata. O nó com chave 26 é cortado de seu pai e transformado em uma raiz não marcada em (d). Ocorre outro corte em cascata, já que o nó com chave 24 também está marcado. Esse nó é cortado de seu pai e transformado em uma raiz não marcada na parte (e). Os cortes em cascata param nesse ponto, já que o nó com chave 7 é uma raiz. (Ainda que esse nó não fosse uma raiz, os cortes em cascata parariam, visto que ele é não marcado.) A parte (e) mostra o resultado da operação FIB-HEAP-DECREASE-KEY, com *H.min* apontando para o novo nó mínimo.

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c .$$

Assim, o custo amortizado de FIB-HEAP-DECREASE-KEY é no máximo

$O(c) + 4 - c = O(1)$,

já que podemos ajustar a escala das unidades de potencial para dominar a constante oculta em $O(c)$.

Agora, você pode ver por que definimos a função potencial para incluir um termo que é duas vezes o número de nós marcados. Quando um nó marcado y é cortado por uma operação de corte em cascata, seu bit de marcação é limpado, o que reduz duas unidades de potencial. Uma unidade de potencial paga o corte e a limpeza do bit de marcação, e a outra unidade compensa o aumento de uma unidade no potencial devido à transformação do nó y em uma raiz.

Eliminando um nó

O pseudocódigo a seguir elimina um nó de um heap de Fibonacci de n nós no tempo amortizado $O(D(n))$. Supomos que não há nenhum valor de chave $-\infty$ no heap de Fibonacci.

```
FIB-HEAP-DELETE(H, x)
1        FIB-HEAP-DECREASE-KEY(H, x, −∞)
2        FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE faz de x o nó mínimo no heap de Fibonacci dando a ele uma chave exclusiva pequena de $-\infty$. Então, o procedimento FIB-HEAP-EXTRACT-MIN remove o nó x do heap de Fibonacci. O tempo amortizado de FIB-HEAP-DELETE é a soma do tempo amortizado $O(1)$ de FIB-HEAP-DECREASE-KEY com o tempo amortizado $O(D(n))$ de FIB-HEAP-EXTRACT-MIN. Como veremos na Seção 19.4 que $D(n) = O(\lg n)$, o tempo amortizado de FIB-HEAP-DELETE é $O(\lg n)$.

Exercícios

19.3-1 Suponha que uma raiz x em um heap de Fibonacci esteja marcada. Explique como x se tornou uma raiz marcada. Demonstre que, para a análise, não importa que x esteja marcada, ainda que ela não seja uma raiz que primeiro foi ligada a outro nó e depois perdeu um filho.

19.3-2 Justifique o tempo amortizado $O(1)$ de FIB-HEAP-DECREASE-KEY como um custo médio por operação, usando análise agregada.

19.4 LIMITANDO O GRAU MÁXIMO

Para provar que o tempo amortizado de FIB-HEAP-EXTRACT-MIN e FIB-HEAP-DELETE é $O(\lg n)$, devemos mostrar que o limite superior $D(n)$ para o grau de qualquer nó de um heap de Fibonacci de n nós é $O(\lg n)$. Em particular, mostraremos que $D(n) \le \log n$, onde é a razão áurea, definida na equação (3.24) como

$$\phi = (1 + \sqrt{5}) / 2 = 1{,}61803\ldots.$$

A chave para a análise é dada a seguir. Para cada nó x dentro de um heap de Fibonacci, defina tamanho(x) como o número de nós, incluindo o próprio x, na subárvore com raiz em x. (Observe que x não precisa estar na lista de raízes — ele pode ser absolutamente qualquer nó.) Mostraremos que tamanho(x) é exponencial em relação a $x.grau$. Lembre-se de que $x.grau$ é sempre mantido como uma medida precisa do grau de x.

Lema 19.1

Seja x qualquer nó em um heap de Fibonacci, e suponha que $x.grau = k$. Seja $y_1, y_2, ..., y_k$ a série de filhos de x na ordem em que eles foram ligados a x, desde o mais antigo até o mais recente. Então, $y_1.grau \geq 0$ e $y_i.grau \geq i - 2$ para $i = 2, 3, ..., k$.

Prova Obviamente, $y_1.grau \geq 0$.
Para $i \geq 2$, observamos que, quando y_i foi ligado a x, todos os $y_1, y_2, ..., y_{i-1}$ eram filhos de x e, assim, também devemos ter tido $x.grau \geq i - 1$. Como o nó y_i é ligado a x (por CONSOLIDATE) somente se $x.grau = y_i.grau$, devemos ter tido também $y_i.grau \geq i - 1$ naquele momento. Desde então, o nó y_i perdeu no máximo um filho, já que ele teria sido cortado de x (por CASCADING-CUT) se tivesse perdido dois filhos. Concluímos que $y_i.grau \geq i - 2$.

Por fim, chegamos à parte da análise que explica o nome "heaps de Fibonacci". Lembre-se de que, na Seção 3.2, vimos que, para $k = 0, 1, 2, ...,$ o k-ésimo número de Fibonacci é definido pela recorrência

$$F_k = \begin{cases} 0 & \text{se } k = 0, \\ 1 & \text{se } k = 1, \\ F_{k-1} + F_{k-2} & \text{se } k \geq 2. \end{cases}$$

O lema a seguir dá outro jeito para expressar F_k.

Lema 19.2

Para todos os inteiros $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

Prova A prova é por indução em relação a k. Quando $k = 0$,

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= F_2. \end{aligned}$$

Agora, consideramos a hipótese indutiva de que $F_k + 1 = 1 + \sum_{i=0}^{k-1} F_i$

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i. \end{aligned}$$

Lema 19.3

Para todos os inteiros $k \geq 0$, o $(k + 2)$-ésimo número de Fibonacci satisfaz $F_k+2 \geq k$.

Prova A prova é por indução em relação a k. Os casos-base são para $k = 0$ e $k = 1$. Quando $k = 0$ temos $F_2 = 1 = {}_0$, e quando $k = 1$ temos $F_3 = 2 > 1{,}619 > {}^1$. A etapa indutiva é para $k \geq 2$, e consideramos que $F_i{+2} > {}_i$ para $i = 0,1, ...,$ k-1. Lembre-se de que é a raiz positiva da equação (3.23), $x_2 = x + 1$.

Assim, temos

$$
\begin{aligned}
F_{k+2} &= F_{k+1} + F_k \\
&\geq \phi^{k-1} + \phi^{k-2} \text{ (pela hipótese de indução)} \\
&= \phi^{k-2}(\phi + 1) \\
&= \phi^{k-2} \cdot \phi^2 \text{ (pela equação (3.23))} \\
&= \phi^k.
\end{aligned}
$$

O lema a seguir e seu corolário concluem a análise.

Lema 19.4

Seja x um nó em um heap de Fibonacci e seja $k = x.grau$. Então, tamanho(x) $\geq F_k + 2 \geq$, onde $\varphi = (1+ \sqrt{5})/ 2$.

Prova Seja s_k o tamanho mínimo possível de um nó de grau k em qualquer heap de Fibonacci. Trivialmente, $s_0 = 1$ e $s_1 = 2$. O número s_k é, no máximo, tamanho(x), e, como adicionar filhos a um nó não pode reduzir o tamanho do nó, o valor de s_k aumenta monotonicamente com k. Considere algum nó z, em qualquer heap de Fibonacci, tal que $z.grau = k$ e tamanho(z) = s_k. Como $s_k \leq$ tamanho(x), calculamos um limite inferior para tamanho(x) calculando um limite inferior para s_k. Como no Lema 19.1, sejam $y_1, y_2, \ldots , y_k$ os filhos de z na ordem em que foram ligados a z. Para limitar s_k, contamos um para o próprio z e um para o primeiro filho y_1 (para o qual tamanho (y_1) ≥ 1), o que dá

$$
\begin{aligned}
\text{tamanho}(x) &\geq s_k \\
&\geq 2 + \sum_{i=2}^{k} s_{y_i \cdot grau} \\
&\geq 2 + \sum_{i=2}^{k} s_{i-2},
\end{aligned}
$$

onde a última linha decorre do Lema 19.1 (de modo que $y_i.grau \geq i$ - 2) e da monotonicidade de s_k (de modo que $s_{y\,.grau} \geq s_i$-2).

Agora, mostramos por indução a k que $s_k \geq F_k + 2$ para todos os inteiros não negativos k. Os casos-base, para $k = 0$ e $k = 1$ são triviais. Para a etapa indutiva, consideramos que $k \geq 2$ e que $s_i \geq F_i + 2$ para $i = 0, 1, ..., k$ - 1. Temos

$$
\begin{aligned}
s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\
&\geq 2 + \sum_{i=2}^{k} F_i \\
&= 1 + \sum_{i=0}^{k} F_i \\
&= F_{k+2} && \text{(pelo Lema 19.2)} \\
&= \phi^k && \text{(pelo Lema 19.3).}
\end{aligned}
$$

Assim, mostramos que tamanho(x) $\geq s_k \geq F_k + 2 \geq$.

Corolário 19.5

O grau máximo $D(n)$ de qualquer nó em um heap de Fibonacci de n nós é $O(\lg n)$.

Prova Seja x um nó em um heap de Fibonacci de n nós e seja $k = x.grau$. Pelo Lema 19.4, temos $n \geq$ tamanho$(x) \geq {}^{k}$. Tomando logaritmos-base, temos $k \leq \log n$. (De fato, como k é um inteiro, $k \leq \log_{ø} n$.) Assim, o grau máximo $D(n)$ de qualquer nó é $O(\lg n)$.

Exercícios

19.4-1 O professor Pinóquio afirma que a altura de um heap de Fibonacci de n nós é $O(\lg n)$. Mostre que o professor está equivocado exibindo, para qualquer inteiro positivo n, uma sequência de operações de heaps de Fibonacci que cria um heap de Fibonacci consistindo em apenas uma árvore que é uma cadeia linear de n nós.

19.4-2 Suponha que generalizamos a regra do corte em cascata para cortar um nó x de seu pai logo que ele perde seu k-ésimo filho, para alguma constante inteira k. (A regra da Seção 19.3 usa $k = 2$.) Para quais valores de k $D(n) = O(\lg n)$?

Problemas

19-1 Implementação alternativa da eliminação

O professor Pisano propôs a seguinte variante do procedimento FIB-HEAP-DELETE, afirmando que ele é executado com maior rapidez quando o nó que está sendo eliminado não é o nó apontado por $H.min$.

```
PISANO-DELETE(H, x)
1 if x = H.min
2         FIB-HEAP-EXTRACT-MIN(H)
3 else y = x.p
4         if y ≠ NIL
5                 CUT(H, x, y)
6                 CASCADING-CUT(H, y)
7         adicionar a lista de filhos de x à lista de raízes de H
8         remover x da lista de raízes de H
```

a. A afirmação do professor de que esse procedimento é executado mais rapidamente se baseia em parte na hipótese de que a linha 7 pode ser executada no tempo real $O(1)$. O que está errado com essa hipótese?

b. Dê um bom limite superior para o tempo real de PISANO-DELETE quando x não é $H.min$. Seu limite deve ser expresso em termos de $x.grau$ e do número c de chamadas ao procedimento CASCADING-CUT.

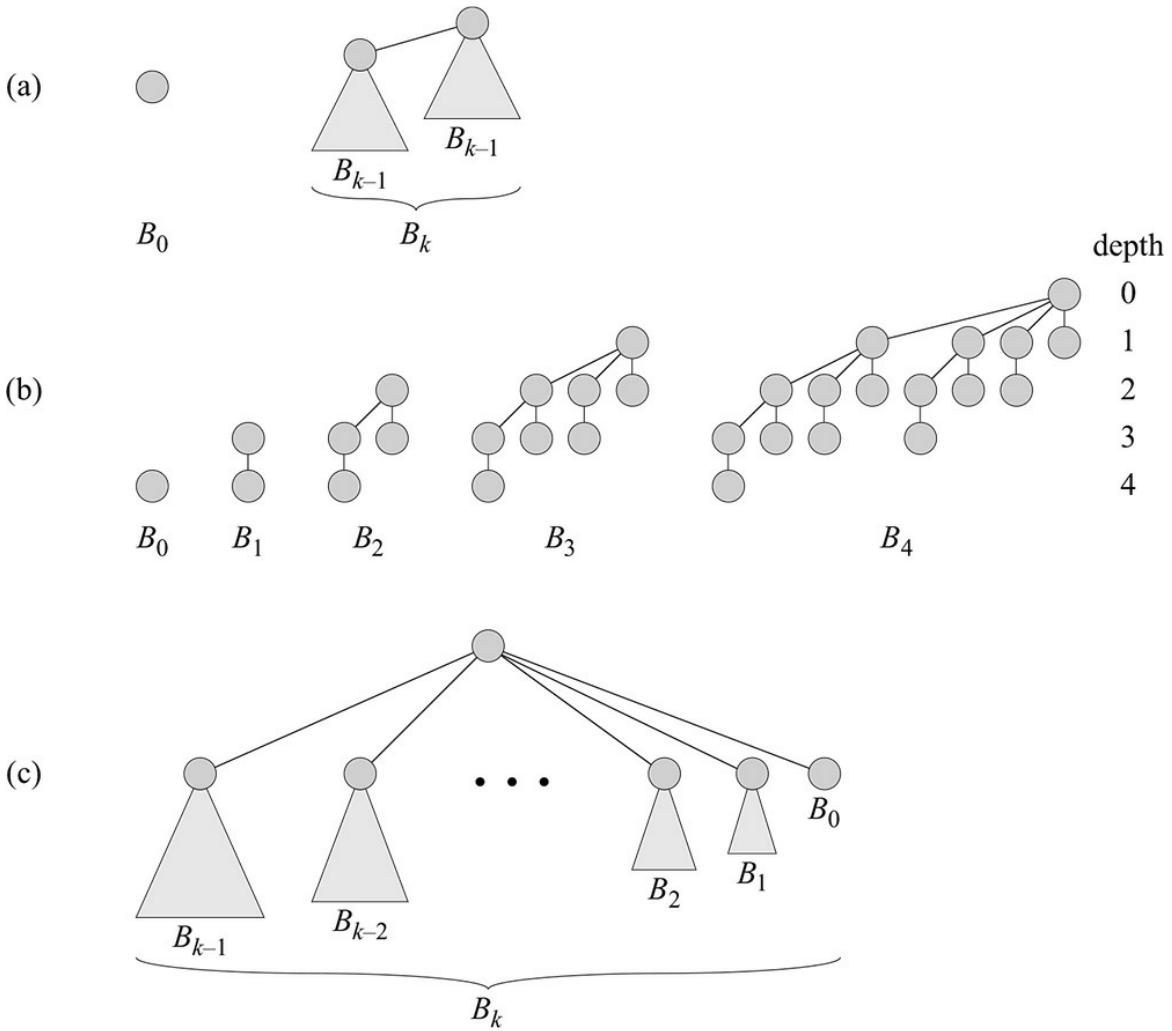
c. Suponha que chamamos PISANO-DELETE(H, x), e seja H' o heap de Fibonacci resultante. Considerando que o nó x não é uma raiz, limite o potencial de H' em termos de $x.grau$, c, $t(H)$ e $m(H)$.

d. Conclua que o tempo amortizado de PISANO-DELETE não é assintoticamente melhor que o de FIB-HEAP-DELETE, mesmo quando $x \neq H.min$.

19-2 Árvores binomiais e heaps binomiais

A ***árvore binomial*** B_k é uma árvore ordenada (veja Seção B.5.2) definida recursivamente. Como mostra a Figura 19.6(a), a árvore binomial B_0 consiste em um único nó. A árvore binomial B_k consiste em duas árvores

binomiais B_{k-1} que são ***ligadas*** uma à outra de modo que a raiz de uma é o filho mais à esquerda da raiz da outra. A Figura 19.6(b) mostra as árvores binomiais de B_0 a B_4.

a. Mostre que, para a árvore binomial B_k ,

1. existem 2^k nós,

2. a altura da árvore é k,

3. existem exatamente $\binom{k}{i}$ nós na profundidade i para $i = 0, 1, \ldots, k$, e

4. a raiz tem grau k, que é maior do que o de qualquer outro nó; além do mais, como a Figura 19.6(c) mostra, se numerarmos os filhos da raiz da esquerda para a direita por $k - 1, k - 2, \ldots, 0$, o filho i será a raiz da subárvore B_i.

Um ***heap binomial*** H é um conjunto de árvores binomiais que satisfaz as seguintes propriedades:

1. Cada nó tem uma *chave* (como um heap de Fibonacci).

2. Cada árvore binomial em H obedece à propriedade de heap de mínimo.

3. Para qualquer inteiro não negativo k, há no máximo uma árvore binomial em H cuja raiz tem grau k.

b. Suponha que um heap binomial H tenha um total de n nós. Discuta a relação entre as árvores binomiais que H contém e a representação binária de n. Conclua que H consiste no máximo em $\lg n + 1$ árvores binomiais.

Suponha que representemos um heap binomial da maneira descrita a seguir. O esquema do filho à esquerda, irmão à direita da Seção 10.4, representa cada árvore binomial dentro de um heap binomial. Cada nó contém sua chave; ponteiros para seu pai, para seu filho à extrema esquerda e para o irmão imediatamente à sua direita (esses ponteiros são NIL quando adequado); e seu grau (como nos heaps de Fibonacci, o número de filhos que ele tem). As raízes formam uma lista de raízes simplesmente ligada, ordenada pelos graus das raízes (de baixo para o alto) e acessamos o heap binomial por um ponteiro para o primeiro nó da lista de raízes.

Figura 19.6 (**a**) Definição recursiva da árvore binomial B_k. Os triângulos representam subárvores enraizadas. (**b**) As árvores binomiais B_0 a B_4. As profundidades dos nós em B_4 são mostradas. (**c**) Outro modo de examinar a árvore binomial B_k.

c. Complete a descrição de como representar um heap binomial (isto é, nome dos atributos, descreva quando os tributos têm o valor NIL e defina como a lista de raízes é organizada) e mostre como implementar as mesmas sete operações em heaps binomiais como este capítulo as implementou em heaps de Fibonacci. Cada operação deve ser executada no tempo do pior caso $O(\lg n)$, onde n é o número de nós no heap binomial (ou, no caso da operação UNION, nos dois heaps binomiais que estão sendo unidos). A operação MAKE-HEAP deve levar tempo constante.

d. Suponha que tivéssemos de implementar somente as operações de heaps intercaláveis em um heap de Fibonacci (isto é, não implementamos as operações DECREASE-KEY ou DELETE). Quais seriam as semelhanças entre as árvores em um heap de Fibonacci e as árvores em um heap binomial? Quais seriam as diferenças? Mostre que o grau máximo em um heap de Fibonacci de n nós seria no máximo $\lg n$.

e. O professor McGee criou uma nova estrutura de dados baseada em heaps de Fibonacci. Um heap de McGee tem a mesma estrutura de um heap de Fibonacci e suporta apenas as operações de heaps intercaláveis. As implementações das operações são idênticas às de heaps de Fibonacci, exceto que, em sua última etapa, a inserção e a união consolidam a lista de raízes. Quais são os tempos de execução do pior caso das operações em heaps de McGee?

19-3 Outras operações de heaps de Fibonacci

Desejamos ampliar um heap de Fibonacci H para suportar duas novas operações sem mudar o tempo de execução amortizado de quaisquer outras operações de heaps de Fibonacci.

a. A operação FIB-HEAP-CHANGE-KEY(H, x, k) troca a chave do nó x pelo valor k. Dê uma implementação eficiente de FIB-HEAP-CHANGE-KEY e analise o tempo de execução amortizado de sua implementação para os casos nos quais k é maior, menor ou igual à $x.chave$.

b. Dê uma implementação eficiente de FIB-HEAP-PRUNE(H, r), que elimine $q = \min(r, H.n)$ nós de H. Você pode escolher quaisquer nós q para eliminar. Analise o tempo de execução amortizado de sua implementação. (*Sugestão*: Talvez seja necessário modificar a estrutura de dados e a função potencial.)

19-4 Heaps 2-3-4

O Capítulo 18 apresentou a árvore 2-3-4, na qual todo nó interno (exceto, possivelmente, a raiz) tem dois, três ou quatro filhos e todas as folhas têm a mesma profundidade. Neste problema, implementaremos ***heaps 2-3-4***, que suportam as operações de heaps intercaláveis.

Os heaps 2-3-4 são diferentes das árvores 2-3-4 nos seguintes aspectos. Nos heaps 2-3-4, apenas as folhas armazenam chaves, e cada folha x armazena exatamente uma chave no atributo $x.chave$. As chaves nas folhas podem estar em qualquer ordem. Cada nó interno x contém um valor $x.pequeno$ que é igual à menor chave armazenada em qualquer folha na subárvore com raiz em x. A raiz r contém um atributo $r.altura$ que dá a altura da árvore. Finalmente, o projeto prevê que heaps 2-3-4 sejam mantidos na memória principal, de modo que leituras e gravações em disco não são necessárias.

Implemente as seguintes operações de heaps 2-3-4. Cada uma das operações nas partes (a)-(e) deve ser executada no tempo $O(\lg n)$ em um heap 2-3-4 com n elementos. A operação UNION na parte (f) deve ser executada no tempo $O(\lg n)$, onde n é o número de elementos nos dois heaps de entrada.

a. MINIMUM, que retorna um ponteiro para a folha com a menor chave.

b. DECREASE-KEY, que diminui a chave de uma dada folha x para um dado valor $k \leq x.chave$.

c. INSERT, que insere a folha x com a chave k.

d. DELETE, que elimina uma dada folha x.

e. EXTRACT-MIN, que extrai a folha com a menor chave.

f. UNION, que une dois heaps 2-3-4, retorna um único heap 2-3-4 e destrói os heaps de entrada.

NOTAS DO CAPÍTULO

Fredman e Tarjan [114] apresentaram os heaps de Fibonacci em uma artigo que também descrevia a aplicação de heaps de Fibonacci aos problemas de caminhos mais curtos de origem única, caminhos mais curtos de todos os pares, emparelhamento bipartido ponderado e o problema da árvore geradora mínima.

Subsequentemente, Driscoll, Gabow, Shrairman e Tarjan [96] desenvolveram "heaps relaxados" como uma alternativa para os heaps de Fibonacci. Eles criaram duas variedades de heaps relaxados. Uma dá os mesmos limites de tempo amortizado que os heaps de Fibonacci. A outra permite a execução de DECREASE-KEY no tempo do pior caso (não amortizado) $O(1)$ e a execução de EXTRACT-MIN e DELETE no tempo do pior caso $O(\lg n)$. Os heaps relaxados também apresentam algumas vantagens em relação aos heaps de Fibonacci em algoritmos paralelos.

Veja também as notas do Capítulo 6 para outras estruturas de dados que suportam operações DECREASE-KEY rápidas quando a sequência de valores retornada por chamadas EXTRACT-MIN são monotonicamente crescentes com o tempo e os dados são inteiros dentro de uma faixa específica.

---

[1] Como mencionado na introdução da Parte V, nossos heaps intercaláveis padrões são heaps de mínimos intercaláveis e, assim, as operações MINIMUM, EXTRACT-MIN e DECREASE-KEY se aplicam. Alternativamente, poderíamos definir um ***heap de máximo intercalável*** com as operações MAXIMUM, EXTRACT-MAX e INCREASE-KEY.

20 Árvores de van Emde Boas

Em capítulos anteriores, vimos estruturas de dados que suportam as operações de uma fila de prioridades — heaps binários no Capítulo 6, árvores vermelho-preto no Capítulo 13,[1] e heaps de Fibonacci no Capítulo 19. Em cada uma dessas estruturas de dados, no mínimo uma operação importante demorou o tempo $O(\lg n)$, seja do pior caso, seja amortizado. Na verdade, como cada uma dessas estruturas de dados baseia suas decisões em comparação de chaves, o limite inferior $(n \lg n)$ para ordenação na Seção 8.1 nos diz que, no mínimo, uma operação levará o tempo $(\lg n)$. Por quê? Se pudéssemos executar as operações INSERT e EXTRACT-MIN no tempo $o(\lg n)$, poderíamos ordenar n chaves no tempo $o(n \lg n)$ executando em primeiro lugar n operações INSERT, seguidas por n operações EXTRACT-MIN.

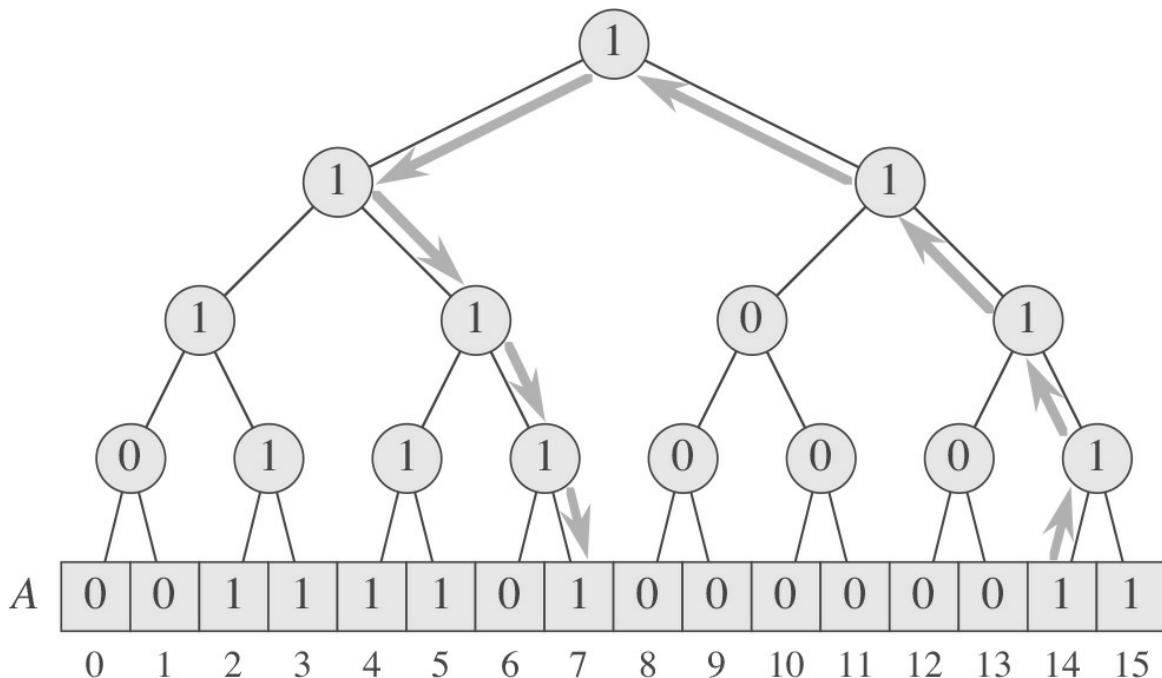
Contudo, no Capítulo 8, vimos que às vezes podemos explorar informações adicionais sobre as chaves para ordenar no tempo $o(n \lg n)$. Em particular, com ordenação por contagem podemos ordenar n chaves, cada uma um inteiro na faixa 0 a k, no tempo $(n + k)$, que é (n) quando $k = O(n)$.

Visto que podemos evitar o limite inferior $(n \lg n)$ para ordenação quando as chaves são inteiros em uma faixa limitada, você bem poderia imaginar se poderíamos executar cada uma das operações de fila de prioridades no tempo $o(\lg n)$ em um cenário semelhante. Veremos, neste capítulo, que podemos: árvores de van Emde Boas suportam as operações de filas de prioridades e algumas outras, cada uma no tempo do pior caso $O(\lg \lg n)$. O senão é que as chaves devem ser inteiros na faixa 0 a $n - 1$ e duplicatas não são permitidas.

Especificamente, árvores de van Emde Boas suportam cada uma das operações de conjunto dinâmico apresentadas na lista da página 230 — SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR — no tempo $O(\lg \lg n)$. Neste capítulo, omitiremos a discussão de dados satélites e focalizaremos somente armazenamento de chaves. Como nos concentramos em chaves e não permitimos o armazenamento de chaves duplicadas, em vez de descrever a operação SEARCH implementaremos a operação mais simples MEMBER (S, x), que retorna um booleano que indica se o valor x está atualmente no conjunto dinâmico S.

Até aqui, usamos o parâmetro n para duas finalidades distintas: o número de elementos no conjunto dinâmico e a faixa dos valores possíveis. Para evitar qualquer confusão, daqui em diante usaremos n para denotar o número de elementos atualmente no conjunto e u para a faixa de valores possíveis, de modo que cada operação de árvore de van Emde é executada no tempo $O(\lg \lg u)$. Denominamos o conjunto $\{0, 1, 2, ..., u - 1\}$ ***universo de*** valores que podem ser armazenados e u o ***tamanho do universo***. Em todo este capítulo, supomos que u é uma potência exata de 2, isto é, $u = 2^k$ para algum inteiro $k \geq 1$.

A Seção 20.1 começa examinando algumas abordagens simples que nos conduzirão na direção certa. Aprimoramos essas abordagens na seção 20.2 introduzindo estruturas proto-van Emde Boas que são recursivas, mas não cumprem nossa meta de operações no tempo $O(\lg \lg u)$. A Seção 20.3 modifica as estruturas proto-van Emde Boas para desenvolver árvores de van Emde Boas e mostra como implementar cada operação no tempo $O(\lg \lg u)$.

20.1 Abordagens preliminares

Nesta seção, examinaremos várias abordagens para armazenar um conjunto dinâmico. Embora nenhuma alcance os limites de tempo $O(\lg \lg u)$ que desejamos, entenderemos um pouco mais sobre árvores de van Emde Boas, o que nos ajudará mais adiante neste capítulo.

Endereçamento direto

Como vimos na Seção 11.1, o endereçamento direto nos dá a abordagem mais simples para armazenar um conjunto dinâmico. Visto que neste capítulo nos preocupamos somente com armazenamento de chaves, podemos simplificar a abordagem do endereçamento direto para armazenar o conjunto dinâmico como um vetor de bits, como discutimos no Exercício 11.1-2. Para armazenar um conjunto dinâmico de valores do universo {0, 1, 2,,..., u - 1}, mantemos um arranjo A[0. . u - 1] de u bits. A entrada $A[x]$ contém um 1 se o valor x estiver no conjunto dinâmico, caso contrário contém um 0. Embora possamos executar cada uma das operações INSERT, DELETE e MEMBER no tempo O (1) com um vetor de bits, cada uma das operações restantes — MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR — leva tempo (u) no pior caso porque poderíamos ter de varrer (u) elementos.[2] Por exemplo, se um conjunto contém somente os valores 0 e u - 1, para encontrar o sucessor de 0 teríamos de varrer as entradas 1 a u - 2 antes de encontrar um 1 em A [u - 1].

Sobrepondo uma estrutura de árvore binária

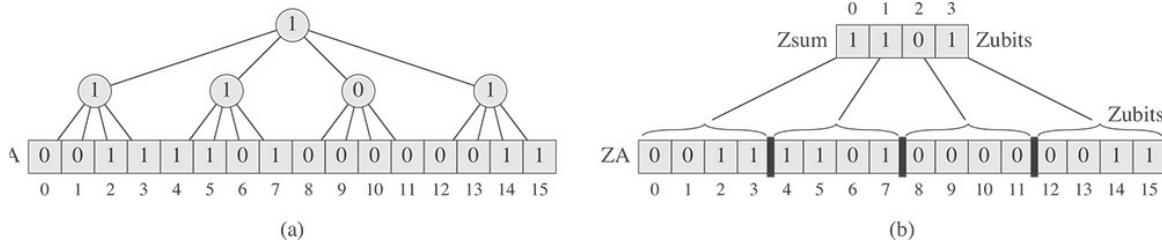
Podemos abreviar longas varreduras no vetor de bits sobrepondo a ele uma árvore binária de bits. A Figura 20.1 mostra um exemplo. As entradas do vetor de bits formam as folhas da árvore binária, e cada nó interno contém um 1 se e somente se qualquer folha em sua subárvore contém um 1. Em outras palavras, o bit armazenado em um nó interno é o OR (OU) lógico de seus dois filhos.

Figura 20.1 A árvore binária de bits sobreposta a um vetor de bits que representa o conjunto {2; 3; 4; 5; 7; 14; 15} quando u = 16. Cada nó interno contém um 1 se e somente se alguma folha em sua subárvore contém um 1. As setas mostram o caminho seguido para determinar o predecessor de 14 no conjunto.

As operações que levaram o tempo (u) do pior caso com um vetor de bits não adornado agora usam a estrutura de árvore:

- Para encontrar o valor mínimo no conjunto, comece na raiz e dirija-se para baixo até as folhas, sempre tomando o nó à extrema esquerda que contém um 1.
- Para encontrar o valor máximo no conjunto, comece na raiz e dirija-se para baixo até as folhas, tomando sempre o nó à extrema direita que contém um 1.
- Para encontrar o sucessor de x, comece na folha indexada por x e prossiga para cima na direção da raiz até entrar em um nó pela esquerda e esse nó tiver um 1 em seu filho à direita z. Então, desça pelo nó z, tomando sempre o nó à extrema esquerda que contém um 1 (isto é, ache o valor mínimo na subárvore com raiz no filho à direita z).
- Para encontrar o predecessor de x, comece na folha indexada por x e continue para cima na direção da raiz até entrar em um nó pela direita e esse nó tiver um 1 em seu filho à esquerda z. Então, desça pelo nó z, tomando sempre o nó à extrema direita que contém um 1 (isto é, ache o valor máximo na subárvore com raiz no filho à esquerda z).

A Figura 20.1 mostra o caminho percorrido para encontrar o predecessor, 7, do valor 14.

Também aumentamos as operações INSERT e DELETE adequadamente. Quando inserimos um valor, armazenamos um 1 em cada nó no caminho simples ascendente da folha adequada até a raiz. Quando eliminamos um valor, percorremos o caminho ascendente da folha adequada até a raiz, recalculando o bit em cada nó interno no caminho como o OU (OR) lógico de seus dois filhos. Visto que a altura da árvore é lg u e cada uma das operações acima executa no máximo uma passagem ascendente pela árvore e no máximo uma passagem descendente, cada operação demora o tempo $O(\lg u)$ no pior caso.

Essa abordagem é apenas marginalmente melhor do que simplesmente usar uma árvore vermelho-preto. Ainda podemos executar a operação MEMBER no tempo $O(1)$, ao passo que executar uma busca em uma árvore vermelho-preto demora o tempo $O(\lg n)$. Então, novamente, se o número n de elementos armazenados é muito menor que o tamanho u do universo, a árvore vermelho-preto será mais rápida para todas as outras operações.

Sobrepondo uma árvore de altura constante

O que acontece se sobrepusermos uma árvore com grau maior? Vamos supor que o tamanho do universo seja $u = 2^{2k}$ para algum inteiro k, de modo que $\sqrt{u}$ é um inteiro. Em vez de sobrepor uma árvore binária ao vetor de bits, sobrepomos uma árvore de grau $\sqrt{u}$. A Figura 20.2(a) mostra tal árvore para o mesmo vetor de bits da Figura 20.1. A altura da árvore resultante é sempre 2.

Como antes, cada nó interno armazena o OR (OU) lógico dos bits dentro de sua subárvore, de modo que os $\sqrt{u}$ nós internos na profundidade 1 resumem cada grupo de $\sqrt{u}$ valores. Como a Figura 20.2(b) demonstra, podemos imaginar esses nós como um arranjo resumo, *summary* $[0..\sqrt{u}-1]$, onde *summary* $[i]$ contém um 1 se e somente se o subarranjo $A\,[i\sqrt{u}..(i+1)\sqrt{u}-1]]$ contém um 1. Denominamos esse subarranjo de $\sqrt{u}$ bits de A o i-ésimo ***cluster*** (grupo). Para um dado valor de x, o bit $A[x]$ aparece no grupo número x / $\sqrt{u}$. Agora, INSERT torna-se uma operação de tempo O(1) para inserir x, defina $A[x]$ e *summaryx* / $\sqrt{u}$ como 1. Podemos usar o arranjo *summary* para executar cada uma das operações MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR e DELETE no tempo O($\sqrt{u}$}:

- Para encontrar o valor mínimo (máximo), encontre a entrada à extrema esquerda (à extrema direita) em *summary* que contém um 1, digamos *summaryi* e depois execute uma busca linear dentro do i-ésimo grupo para o 1 à extrema esquerda (à extrema direita).

Figura 20.2 (a) Árvore de grau $\sqrt{u}$ sobreposta ao mesmo vetor de bits da Figura 20.1. Cada nó interno armazena o OU (OR) lógico dos bits em sua subárvore. **(b)** Uma vista da mesma estrutura, porém na qual os nós >internos na profundidade 1 são tratados como um arranjo *summary* $0..\sqrt{u}-1$, onde *summary* i é o OU (OR) lógico do subarranjo $A[i\sqrt{u}..(i+1)\sqrt{u}-1]$.

- Para encontrar o sucessor (predecessor) de x, primeiro execute uma busca à direita (à esquerda) dentro de seu grupo. Se encontrar um 1, essa posição dá o resultado. Caso contrário, seja $i = x / \sqrt{u}$ e execute uma busca à direita (à esquerda) dentro do arranjo *summary* partindo do índice i. A primeira posição que contém um 1 dá o índice de um grupo. Execute uma busca dentro desse grupo para o 1 à extrema esquerda (à extrema direita). Essa posição contém o sucessor (predecessor).
- Para eliminar o valor x, seja $i = x / \sqrt{u}$. Defina A[x] como 0 e *summary*[i] como o OU (OR) lógico dos bits no i-ésimo grupo.

Em cada uma dessas operações, executamos a busca em, no máximo, dois grupos de $\sqrt{u}$ bits, mais o arranjo *summary* e, assim, cada operação leva o tempo $O(\sqrt{u})$.

À primeira vista, parece que nosso progresso foi negativo. Sobrepor a árvore binária nos deu operações de tempo O(lg u), que são assintoticamente mais rápidas do que as de tempo $O(\sqrt{u})$. Contudo, veremos que usar uma árvore de grau $\sqrt{u}$ é uma ideia fundamental das árvores de van Emde Boas. Continuaremos nesse caminho na próxima seção.

Exercícios

20.1-1 Modifique as estruturas de dados nesta seção para suportar chaves duplicadas.

20.1-2 Modifique as estruturas de dados nesta seção para suportar chaves que têm dados satélites associados.

20.1-3 Observe que, usando as estruturas nesta seção, o modo como encontramos o sucessor e o predecessor de um valor x não depende de x estar no conjunto no momento em questão. Mostre como encontrar o sucessor de x em uma árvore de busca binária quando x não está armazenado na árvore.

20.1-4 Suponha que, em vez de sobrepor uma árvore de grau $\sqrt{u}$, quiséssemos sobrepor uma árvore de grau $u_{1/k}$, onde $k > 1$ é uma constante. Qual seria a altura de tal árvore e quanto tempo demoraria cada uma das operações?

20.2 UMA ESTRUTURA RECURSIVA

Nesta seção, modificamos a ideia de sobrepor uma árvore de grau $\sqrt{u}$ a um vetor de bits. Na seção anterior, usamos uma estrutura de resumo de tamanho $\sqrt{u}$, na qual cada entrada aponta para uma outra estrutura de tamanho $\sqrt{u}$. Agora, tornamos a estrutura recursiva, reduzindo o tamanho do universo pela raiz quadrada em cada nível de recursão. Começando com um universo de tamanho u, fazemos estruturas que contêm $\sqrt{u} = u_{1/2}$ itens que, por sua vez, contêm estruturas de $u_{1/4}$ itens, que contêm estruturas de $u_{1/8}$ itens, e assim por diante, até um tamanho-base de 2.

Por simplicidade, nesta seção consideramos que $u = 2^2$ para algum inteiro k, de modo que u, $u_{1/2}$, $u_{1/4}$,...sejam inteiros. Essa restrição seria bem rigorosa na prática e permitiria somente valores de u na sequência 2, 4, 16, 256,

65536,.... Na próxima seção, veremos como relaxar essa condição para $u = 2^k$ para algum inteiro k. Visto que a estrutura que examinamos nesta seção é apenas uma precursora da verdadeira estrutura de árvore de van Emde Boas, toleramos essa restrição em favor de um melhor entendimento.

Lembrando que nossa meta é conseguir tempos de execução de $O(\lg \lg u)$ para as operações, vamos pensar em como poderíamos obter tais tempos de execução. No final da Seção 4.3 vimos que, trocando variáveis, podíamos mostrar que a recorrência

$$T(n) = 2T\left(\left\lfloor \sqrt{n} \right\rfloor\right) + \lg n \qquad (20.1)$$

tem a solução $T(n) = O(\lg n \lg \lg n)$. Vamos considerar ume recorrência semelhante, porém mais simples:

$$T(u) = T\left(\sqrt{u}\right) + O(1). \qquad (20.2)$$

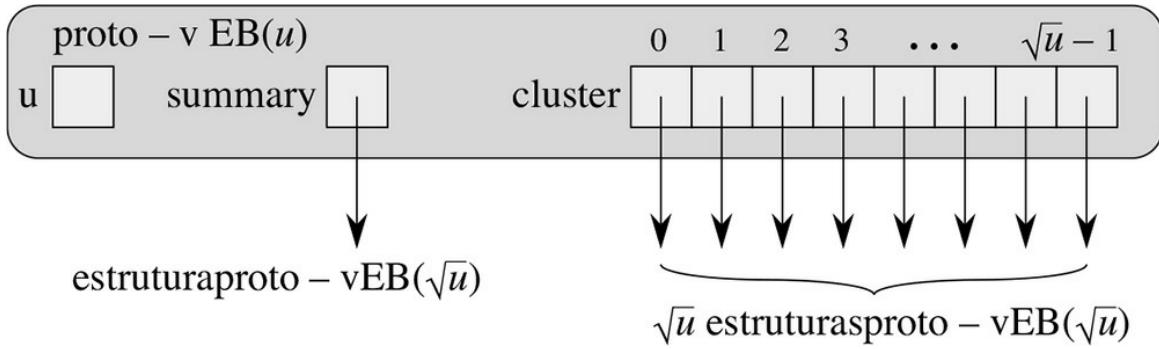
Se usarmos a mesma técnica, troca de variáveis, podemos mostrar que a recorrência (20.2) tem solução $T(u) = O(\lg \lg u)$. Seja $m = \lg u$, de modo que $u = 2^m$ e temos

$$T(2^m) = T(2^{m/2}) + O(1)$$

Agora, renomeamos $S(m) = T(2^m)$, o que dá a nova recorrência

$$S(m) = S(m/2) + O(1).$$

Pelo caso 2 do método mestre, essa recorrência tem a solução $S(m) = O(\lg m)$. Voltamos a trocar $S(m)$ por $T(u)$, o que dá $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u)$.

A recorrência (20.2) guiará a nossa busca por uma estrutura de dados. Projetaremos uma estrutura de dados recursiva que é reduzida por um fator de $\sqrt{u}$ em cada nível de sua recursão.

Quando uma operação percorre essa estrutura de dados, gasta uma quantidade de tempo constante em cada nível antes de executar a recursão para o nível abaixo. Então, a recorrência (20.2) caracteriza o tempo de execução da operação.

Apresentamos agora, um outro modo de pensar sobre como o termo $\lg \lg u$ acaba aparecendo na solução da recorrência (20.2). Examinando o tamanho do universo em cada nível da estrutura de dados recursiva, vemos a sequência u, $u^{1/2}$, $u^{1/4}$, $u^{1/8}$,... Se considerarmos quantos bits precisamos para armazenar o tamanho de universo em cada nível, precisamos de $\lg u$ no nível superior, e cada nível precisa de metade dos bits do nível anterior. Em geral, se começarmos com b bits e dividirmos o número de bits pela metade em cada nível, após $\lg b$ níveis, chegaremos a apenas um bit. Como $b = \lg u$, vemos que, após $\lg \lg u$ níveis, temos um tamanho de universo igual a 2.

Voltando a examinar a estrutura de dados na Figura 20.2, determinado valor x reside no grupo número $x / \sqrt{u}$. Se virmos x como um inteiro binário de $\lg u$ bits, aquele número de grupo, $x / \sqrt{u}$, é dado pelos $(\lg u)/2$ bits mais significativos de x. Dentro de seu grupo, x aparece na posição $x \bmod \sqrt{u}$, que é dada pelos $(\lg u)/2$ bits menos significativos de x. Precisaremos indexar desse modo e, portanto, vamos definir algumas funções que nos ajudarão a fazer isso.

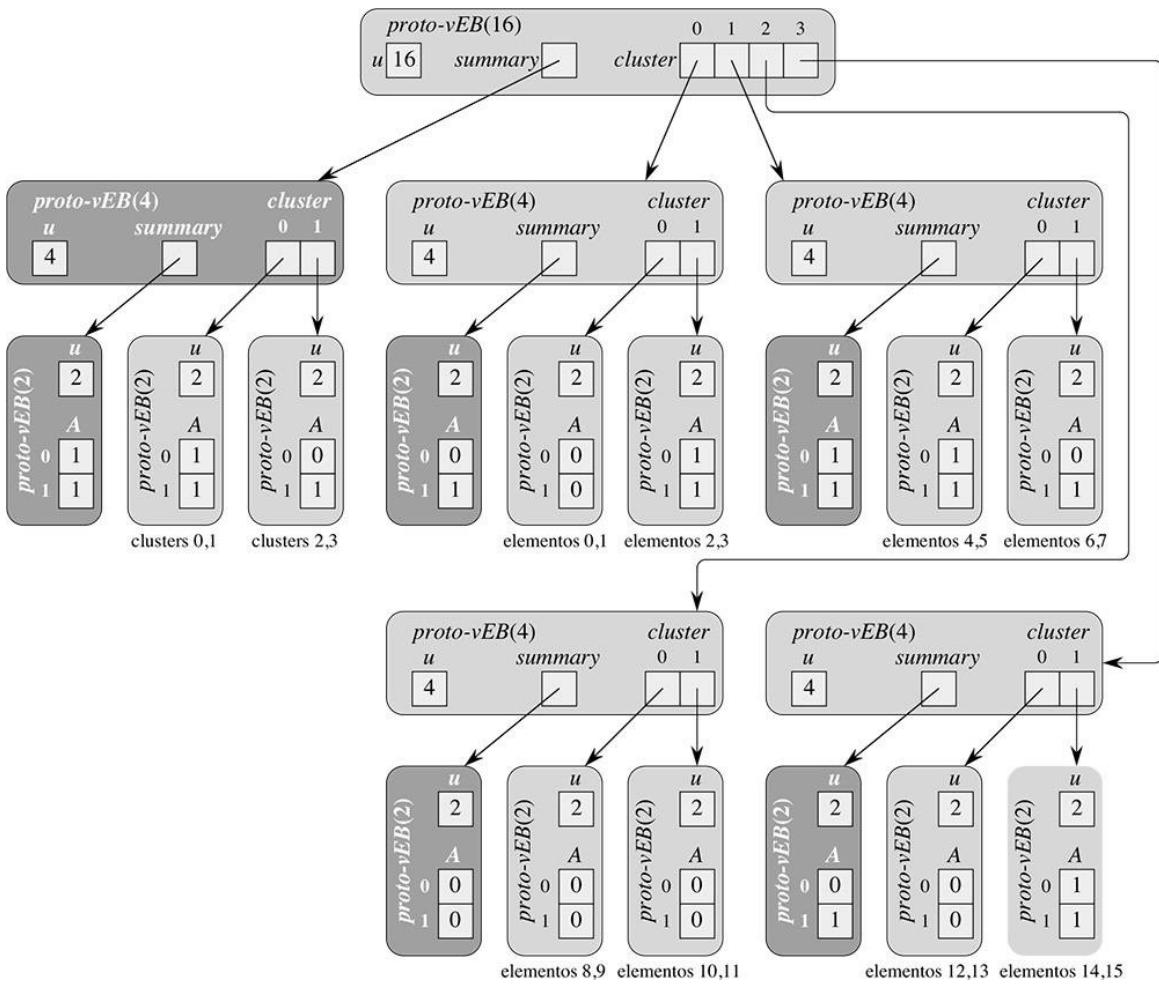
$$\begin{aligned} high(x) &= \left\lfloor x / \sqrt{u} \right\rfloor, \\ low(x) &= x \bmod \sqrt{u}, \\ index(x, y) &= x\sqrt{u} + y. \end{aligned}$$

Figura 20.3 A informação em uma estrutura proto-v*EB*(u) quando $u \geq 4$. A estrutura contém o tamanho de universo u, um ponteiro *summary* para uma estrutura proto-v*EB* ($\sqrt{u}$) e um arranjo *cluster* 0.. $\sqrt{u}-1$ de ponteiros $\sqrt{u}$ para estruturas proto-v*EB* ($\sqrt{u}$) .

A função high(x) dá os (lg u)/2 bits mais significativos de x, produzindo o número do grupo de x. A função low(x) dá os (lg u)/2 bits menos significativos de x e a posição de x dentro de seu grupo. A função index$x$(y) constrói um número de elemento a partir de x e y, tratando x como os (lg u)/2 bits mais significativos do número do elemento e y como os (lg u)/2 bits menos significativos. Temos a identidade x = index(high(x), low(x)). O valor de u usado por cada uma dessas funções será sempre o tamanho de universo da estrutura de dados na qual chamamos a função, que muda ao descermos e entrarmos na estrutura recursiva.

20.2.1 ESTRUTURAS PROTO-VAN EMDE BOAS

De acordo com o que indica a recorrência (20.2), vamos projetar uma estrutura de dados recursiva para suportar as operações. Apesar de que essa estrutura de dados não conseguirá cumprir nossa meta de tempo O(lg lg u) para algumas operações, ela serve como base para a estrutura da árvore de van Emde Boas que veremos na Seção 20.3.

Para o universo {0, 1, 2,..., u - 1}, definimos uma estrutura ***proto-van Emde Boas (***ou estrutura ***proto-vEB***, que denotamos por *proto-vEB*(u) recursivamente, da maneira descrita a seguir. Cada *estrutura proto-vEB*(u) contém um atributo u que dá seu tamanho de universo. Além disso, ela contém o seguinte:

- Se $u = 2$, então ela é o tamanho-base e contém um arranjo A0. .1 de dois bits.
- Caso contrário, $u = 2^{2k}$ para algum inteiro $k \geq 1$, de modo que $u \geq 4$. Além do tamanho de universo u, a estrutura de dados *proto-vEB*(u) contém os seguintes atributos, ilustrados na Figura 20.3:
  - um ponteiro denominado *summary* para uma estrutura *proto-vEB* ($\sqrt{u}$) (resumo) e
  - um arranjo *cluster* [0..$\sqrt{u}$–1] de $\sqrt{u}$ ponteiros, cada um para uma *estrutura proto-vEB* ($\sqrt{u}$).

O elemento x, onde $0 \leq x < u$, é armazenado recursivamente no grupo numerado high(x) como elemento low(x) dentro daquele grupo.

Na estrutura de dois níveis da seção anterior, cada nó armazena um arranjo resumo de tamanho $\sqrt{u}$, no qual cada entrada contém um bit. Pelo índice de cada entrada, podemos calcular o índice inicial do subarranjo de tamanho $\sqrt{u}$ que o bit resume. Na estrutura proto-vEB, usamos ponteiros explícitos em vez de cálculos de índices.

O arranjo *summary* contém os bits de resumo armazenados recursivamente em uma estrutura proto-vEB, e o arranjo *cluster* contém $\sqrt{u}$ ponteiros.

A Figura 20.4 mostra uma estrutura *proto-vEB*(16) totalmente expandida, que representa o conjunto {2, 3, 4, 5, 7, 14, 15}. Se o valor i estiver na estrutura proto-vEB para a qual *summary* aponta, então o i-ésimo grupo contém algum valor no conjunto que está sendo representado.

Figura 20.4 Uma estrutura *proto-EB*(16) que representa o conjunto {2, 3, 4, 5, 7, 14, 15}. Ela aponta para quatro estruturas *proto-vEB*(4) em *cluster*0..3, e para uma estrutura resumo, que é também uma *proto-vEB*(4). Cada *estrutura proto-vEB*(4) aponta para duas estruturas *proto-vEB(2)* em *cluster* 0..1, e para uma resumo *proto-vEB*(2). Cada *estrutura proto-vEB(2)* contém apenas um arranjo *A*0..1 de dois bits. As estruturas *proto-EB*(2) *acima de* "elementos *i,j*" armazenam bits *i* e *j* do conjunto dinâmico propriamente dito, e as estruturas *proto-vEB*(2) acima dos "grupos *i,j*" armazenam os bits resumo para os grupos *i* e j na estrutura *proto-vEB*(16) do nível superior. Por questão de clareza, os retângulos sombreados em tom mais escuro indicam o nível superior de uma estrutura proto-vEB que armazena informações de resumo para sua estrutura pai; em todos os outros aspectos, tal estrutura proto-vEB é idêntica a qualquer outra estrutura proto-vEB com o mesmo tamanho de universo.

Como na árvore de altura constante, *cluster*[*i*] representa os valores $i\sqrt{u}$ de $(i + 1)\sqrt{u} - 1$, que forma o *i*-ésimo grupo.

No nível-base, os elementos dos conjuntos dinâmicos propriamente ditos são armazenados em algumas das estruturas *proto-vEB*(2), e as estruturas *proto-vEB*(2) restantes armazenam bits de resumo. Abaixo de cada uma das estruturas bases não *summary*, a figura indica quais bits tal estrutura armazena. Por exemplo, a *estrutura proto-vEB*(2) rotulada "elementos 6,7" armazena o bit 6 (0, visto que o elemento 6 não está no conjunto) em seu A[0]" e o bit 7 (1, visto que o elemento 7 está no conjunto) em seu A[1].

Como os grupos, cada resumo é apenas um conjunto dinâmico com tamanho de universo $\sqrt{u}$, portanto representamos cada resumo como uma estrutura *proto-vEB* ($\sqrt{u}$). Os quatro bits de resumo para a *estrutura proto-vEB*(16) principal estão à extrema esquerda da estrutura *proto-vEB*(4) e, afinal, aparecem em duas estruturas *proto-vEB*(2). Por exemplo, a estrutura *proto-vEB*(2) rotulada "grupos 2,3" tem *A*[0] = 0, o que indica que o grupo 2 da estrutura *proto-vEB(16)* (que contém os elementos 8, 9, 10, 11) é todo 0 e A[1] = 1, que nos diz que o grupo 3 (que contém os elementos 12, 13, 14, 15) tem no mínimo um 1. Cada estrutura *proto-vEB*(4) aponta para seu próprio resumo que, por sua vez, é armazenado como uma estrutura *proto-vEB*(2). Por exemplo, examine a *estrutura proto-*

vEB(2). imediatamente à esquerda da estrutura rotulada "elementos 0,1". Como seu A[0] é 0, ela nos diz que a estrutura "elementos 0, 1 é toda 0, e como seu A[1] é 1, sabemos que a estrutura "elementos 2,3" contém no mínimo um 1.

20.2.2 OPERAÇÕES EM UMA ESTRUTURA PROTO-VAN EMDE BOAS

Agora descreveremos como executar operações em uma estrutura proto-vEB. Em primeiro lugar examinamos as operações de consulta — MEMBER, MINIMUM e SUCCESSOR — que não mudam a estrutura proto-vEB. Em seguida discutiremos INSERT e DELETE. Deixamos MAXIMUM e PREDECESSOR, que são simétricas em relação a MINIMUM e SUCCESSOR, respectivamente, para o Exercício 20.2-1.

Cada uma das operações MEMBER, SUCCESSOR, PREDECESSOR, INSERT e DELETE adota um parâmetro x, juntamente com a estrutura proto-vEB V. Cada uma dessas operações considera que $0 \leq x < V.u$.

Determinando se um valor está no conjunto

Para executar MEMBER(x), precisamos encontrar o bit que corresponde a x dentro da estrutura *proto-vEB*(2) adequada. Podemos fazer isso no tempo $O(\lg \lg u)$, se evitarmos, totalmente as estruturas *summary*. O seguinte procedimento toma uma estrutura *proto-vEB V* e um valor x, e retorna um bit que indica se x está no conjunto dinâmico mantido por V.

```
PROTO-vEB-MEMBER(V, x)
1  if V.u == 2
2       return V.A[x]
3  else return PROTO-vEB-MEMBER.(V.cluster[high(x)], low(x))
```

O procedimento PROTO-vEB-MEMBER funciona da maneira descrita a seguir. A linha 1 testa se estamos em um caso-base, quando V é uma estrutura *proto*-v*EB*(2). A linha 2 trata o caso-base simplesmente retornando o bit de arranjo A adequado. A linha 3 trata o caso recursivo, "perfurando" a estrutura *proto*-*v*EB menor adequada. O valor high(x) informa qual estrutura *proto-vEB*($\sqrt{u}$) visitamos, e low(x) determina qual elemento dentro daquela estrutura *proto-vEB* ($\sqrt{u}$) estamos consultando.

Vamos ver o que acontece quando chamamos PROTO-VEB-MEMBER.V(6) na *estrutura proto-vEB*(16) na Figura 20.4. Visto que high(6) = 1 quando u = 16, executamos recursão na *estrutura proto-vEB*(4) na parte superior à direita e indagamos sobre o elemento low(6) = 2 daquela estrutura. Nessa chamada recursiva, u = 4 e, assim, executamos novamente uma recursão. Com u = 4, temos high(2) = 1 e low(2) = 0 e assim indagamos sobre o elemento 0 da *estrutura proto-vEB*(2) na parte superior à direita. Essa chamada recursiva revela ser um caso-base e, assim, retorna A[0] = 0 de volta para cima por meio da cadeia de chamadas recursivas. Assim, obtemos o resultado de que PROTO-vEB-MEMBER(V, 6) retorna 0, indicando que 6 não está no conjunto.

Para determinar o tempo de execução de PROTO-vEB-MEMBER, seja $T(u)$ seu tempo de execução em uma *estrutura proto-vEB*(u). Cada chamada recursiva leva tempo constante, não incluindo o tempo que demoram as chamadas recursivas que o procedimento faz. Quando PROTO-vEB-MEMBER faz uma chamada recursiva, faz uma chamada em uma estrutura *proto-vEB* ($\sqrt{u}$) Assim, podemos caracterizar o tempo de execução pela recorrência $T(u)$ $T(\sqrt{u})$ $O(1)$, que já vimos como recorrência (20.2). Sua solução é $T(u) = O(\lg \lg u)$ e, portanto, concluímos que PROTO-vEB-MEMBER é executada no tempo $O(\lg \lg u)$.

Achando o elemento mínimo

Agora examinamos como executar a operação MINIMUM. O procedimento PROTO-VEB-MINIMUM(V) retorna o elemento mínimo na estrutura proto-vEB V ou NIL se V representar um conjunto vazio.

```
PROTO-vEB-MINIMUM(V)
1  if V.u == 2
2      if V.A[0] == 1
3          return 0
4      elseif V.A[1] == 1
5          return 1
6      else return NIL
7  else min-cluster = PROTO-vEB-MINIMUM (V.summary)
8      if min-cluster == NIL
9          return NIL
10     else offset = PROTO-vEB-MINIMUM(V.cluster[min-cluster])
11     return index(min-cluster, offset)
```

Esse procedimento funciona da seguinte maneira: a linha 1 testa se é o caso-base, que as linhas 2-6 tratam por força bruta. As linhas 7-11 tratam o caso recursivo. Primeiro, a linha 7 encontra o número do primeiro grupo que contém um elemento do conjunto. Faz isso recursivamente chamando PROTO-vEB-MINIMUM em V.*summary*, que é uma estrutura *proto-vEB*($\sqrt{u}$).

A linha 7 atribui esse número de grupo à variável *min-cluster*. Se o conjunto é vazio, a chamada recursiva retornou NIL, e a linha 9 retorna NIL. Caso contrário, o elemento mínimo do conjunto está em algum lugar no grupo número *min-cluster*. A chamada recursiva na linha 10 encontra o deslocamento (offset) dentro do grupo do elemento mínimo nesse grupo. Finalmente, a linha 11 constrói o valor do elemento mínimo a partir do número do grupo e do deslocamento (offset) e devolve esse valor.

Embora consultar as informações de resumo nos permita encontrar rapidamente o grupo que contém o elemento mínimo, como esse procedimento faz duas chamadas recursivas em estruturas *proto-vEB*($\sqrt{u}$), não é executada no tempo O(lg lg u) no pior caso. Denotando por $T(u)$ o tempo do pior caso para PROTO-vEB-MINIMUM em uma estrutura *proto-vEB*(u), temos a recorrência

$$T(u) = 2T\left(\sqrt{u}\right) + O(1). \tag{20.3}$$

Novamente, usamos uma troca de variáveis para resolver essa recorrência, fazendo $m = \lg u$, o que dá

$$T\ (2^m) = 2T\ (2^{m/2})\ + O(1).$$

Renomeando $S(m) = T\ (2^m)$ temos

$$S(m) = 2S(m/2) + O(1)\ ,$$

que, pelo caso 1 do método mestre, tem a solução $S(m) = (m)$. Destrocando $S(m)$ para $T(u)$, temos que $T(u) = T(2^m) = S(m) = (m) = (\lg u)$. Assim, vemos que, por causa da segunda chamada recursiva, PROTO-vEB-MINIMUM é executada no tempo $(\lg u)$ em vez de no tempo desejado $O(\lg \lg u)$.

Encontrando o sucessor

A operação SUCCESSOR é ainda pior. No pior caso, faz duas chamadas recursivas, juntamente com a chamada a PROTO-vEB-MINIMUM. O procedimento PROTO-vEB-SUCCESSOR (V, x) retorna o menor elemento na estrutura proto-vEB V, que é maior que x ou NIL se nenhum elemento em V é maior que x. O procedimento não exige que x seja um membro do conjunto, mas supõe que $0 \le x < V.u$.

```
PROTO-vEB-SUCCESSOR(V, X)
1  if V.u == 2
2        if x == 0 e V.A[1] == 1
3              return 1
4        else return NIL
5  else offset = PROTO-vEB-SUCCESSOR(V.cluster[high(x)], low(x))
6        if offset ≠ NIL
7              return index(high(x), offset)
8        else succ-cluster = PROTO-vEB-SUCCESSOR(V.summary, high(x))
9              if succ-cluster == NIL
10                   return NIL
11             else offset = PROTO-vEB-MINIMUM(V, cluster[succ-cluster])
12                   return index(succ-cluster, offset)
```

O procedimento PROTO-vEB-SUCCESSOR funciona da maneira descrita a seguir. Como sempre, a linha 1 testa para o caso-base, cujas linhas 2-4 tratam por força bruta: o único modo pelo qual x pode ter um sucessor dentro de uma *estrutura proto-vEB*(2) é quando $x = 0$ e $A[1]$ é 1. As linhas 5-12 tratam o caso recursivo. A linha 5 procura um sucessor para x dentro do grupo de x, designando o resultado a *offset*. A linha 6 determina se x tem um sucessor dentro de seu grupo; se tiver, a linha 7 calcula e retorna o valor desse sucessor. Caso contrário, temos de procurar em outros grupos. A linha 8 designa a *succ-cluster* o número do próximo grupo não vazio, usando as informações de resumo para encontrá-lo. A linha 9 testa se *succ-cluster* é NIL e a linha 10 retorna NIL se todos os grupos subsequentes são vazios. Se *succ-cluster* é não NIL, a linha 11 designa o primeiro elemento dentro daquele grupo a *offset*, e a linha 12 calcula e retorna o elemento mínimo naquele grupo.

No pior caso, PROTO-vEB-SUCCESSOR chama a si mesma recursivamente duas vezes, nas e*struturas proto-vEB*($\sqrt{u}$), e faz uma chamada a PROTO-vEB-MINIMUM em uma estrutura *proto-vEB*($\sqrt{u}$). Assim, a recorrência para o tempo de execução do pior caso $T(u)$ de PROTO-vEB-SUCCESSOR é

$$\begin{aligned} T(u) &= 2T\left(\sqrt{u}\right) + \Theta\left(\lg\sqrt{u}\right) \\ &= 2T\left(\sqrt{u}\right) + \Theta\left(\lg u\right). \end{aligned}$$

Podemos empregar a mesma técnica que usamos para a recorrência (20.1) para mostrar que essa recorrência tem a solução $T(u) = (\lg u \lg \lg u)$. Assim, PROTO-vEB-SUCCESSOR é assintoticamente mais lenta que PROTO-vEB-MINIMUM.

Inserindo um elemento

Para inserir um elemento, precisamos inseri-lo no grupo adequado e também atribuir 1 ao bit de resumo para aquele grupo. O procedimento PROTO-vEB-INSERT (V, x) insere o valor x na estrutura proto-vEB V.

```
PROTO-vEB-INSERT (V, x)
1  if V.u == 2
2     V.A[x] = 1
3  else PROTO-vEB-INSERT(V.cluster [high.x], low.x))
4  PROTO-vEB-INSERT(V.summary, high(x))
```

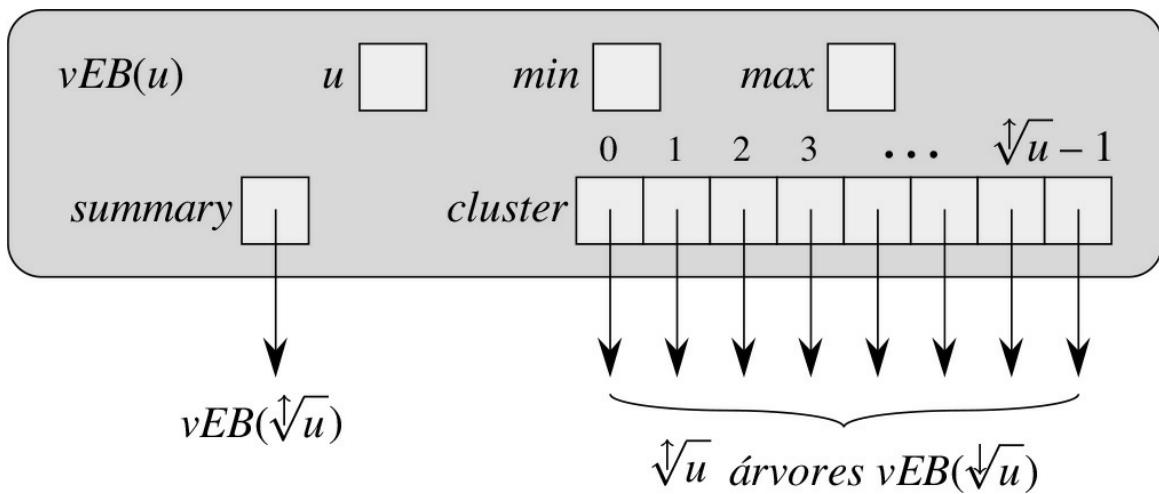
No caso-base, a linha 2 define como 1 o bit adequado no arranjo A. No caso recursivo, a chamada recursiva na linha 3 insere x no grupo adequado, e a linha 4 define como 1 o bit de resumo para aquele grupo.

Como PROTO-vEB-INSERT faz duas chamadas recursivas no pior caso, a recorrência (20.3) caracteriza seu tempo de execução. Por consequência, PROTO-vEB-INSERT é executada no tempo (lg u).

Eliminando um elemento

A operação DELETE é mais complicada que a inserção. Ao passo que sempre podemos definir um bit de resumo como 1 quando executamos uma inserção, nem sempre podemos redefinir o mesmo bit como 0 quando executamos uma eliminação. Precisamos determinar se qualquer bit no grupo adequado é 1. Como definimos estruturas proto-vEB, teríamos de examinar todos os $\sqrt{u}$ bits dentro de um grupo para determinar se qualquer deles é 1. Alternativamente, poderíamos adicionar um atributo n à estrutura proto-vEB para contar quantos elementos ela tem. Deixamos a implementação de PROTO-vEB-DELETE para os Exercícios 20.2-2 e 20.2-3.

É claro que precisamos modificar a estrutura proto-vEB para conseguir que cada operação faça, no máximo, uma chamada recursiva. Na próxima seção veremos como fazer isso.

Exercícios

20.2-1 Escreva pseudocódigo para os procedimentos PROTO-VEB-MAXIMUM e PROTO-VEB-PREDECESSOR.

20.2-2 Escreva pseudocódigo para PROTO-vEB-DELETE. O pseudocódigo deve atualizar o bit de resumo adequado executando uma varredura nos bits relacionados dentro do grupo. Qual é o tempo de execução do pior caso de seu procedimento?

20.2-3 Adicione o atributo n a cada estrutura proto-vEB, para dar o número de elementos atualmente no conjunto que ele representa e escreva pseudocódigo para PROTO-vEB-DELETE que use o atributo n para decidir quando atribuir 0 aos bits de resumo. Qual é o tempo de execução do pior caso do seu procedimento? Quais outros procedimentos precisam mudar por causa do novo atributo? Essas mudanças afetam seus tempos de execução?

20.2-4 Modifique a estrutura proto-vEB para suportar chaves duplicadas.

20.2-5 Modifique a estrutura proto-vEB para suportar chaves que tenham dados satélites associados.

20.2-6 Escreva pseudocódigo para um procedimento que cria uma estrutura *proto-vEB(u)*.

20.2-7 Demonstre que, se a linha 9 de PROTO-vEB-MINIMUM é executada, a estrutura proto-vEB é vazia.

20.2-8 Suponha que projetemos uma estrutura proto-vEB na qual cada arranjo *cluster* tenha somente $u_{1/4}$ elementos. Quais seriam os tempos de execução de cada operação?

20.3 A ÁRVORE DE VAN EMDE BOAS

A estrutura proto-vEB da seção anterior está próxima do que precisamos para conseguir tempos de execução O(lg lg u). Ela não alcança esse objetivo porque temos de executar recursão, muitas vezes, na maioria das operações. Nesta seção, projetaremos uma estrutura de dados que é semelhante à estrutura proto-vEB, mas armazena uma quantidade um pouco maior de informações, o que resulta na eliminação de algumas operações recursivas.

Na Seção 20.2, observamos que a premissa que adotamos em relação ao tamanho de universo — de $u = 2^{2k}$ para algum inteiro k — é indevidamente restritiva, já que confina os valores possíveis de u a um conjunto excessivamente esparso. Portanto, deste ponto em diante permitiremos que o tamanho de universo u seja qualquer potência exata de 2 e, quando $\sqrt{u}$ não for um inteiro — isto é, se u for um potência ímpar de 2 ($u = 2^{2k+1}$ para algum inteiro $k \geq 0$) — então dividiremos os lg u bits de um número em bits mais significativos (lg u)/ 2 e bits menos significativos (lg u)/ 2. Por conveniência, denotamos $2^{\lg u)/2}$ (a "raiz quadrada superior" de u) por $\sqrt[\downarrow]{u}$ e $2^{[(\lg u)/2]}$(a "raiz quadrada inferior" de u) por $\sqrt[\downarrow]{u}$, de modo que $u = \sqrt[\uparrow]{u} \cdot \sqrt[\downarrow]{u}$ e, quando u é uma potência par de 2 ($u = 2^{2k}$ para algum inteiro k), $\sqrt[\uparrow]{u} = \sqrt[\downarrow]{u} = \sqrt{u}$. Como agora permitimos que u seja uma potência ímpar de 2, temos de redefinir nossas prestimosas funções da Seção 20.2:

$$
\begin{aligned}
alto(x) &= \left\lfloor x / \sqrt[\downarrow]{u} \right\rfloor, \\
baixo(x) &= x \bmod \sqrt[\downarrow]{u}, \\
índice(x, y) &= x\sqrt[\downarrow]{u} + y.
\end{aligned}
$$

Figura 20.5 As informações em uma árvore $vEB(u)$ quando u > 2. A estrutura contém o tamanho de universo u, elementos *min* e *max*, um ponteiro *summary* para uma árvore $\sqrt[\uparrow]{u}$ e um arranjo *cluster* 0..$\sqrt[\uparrow]{u}$ −1 de $\sqrt[\uparrow]{u}$ ponteiros para árvores *vEB* ($\sqrt[\downarrow]{u}$).

20.3.1 ÁRVORES DE VAN EMDE BOAS

A ***árvore de van Emde Boas***, ou ***árvore vEB***, modifica a estrutura proto-vEB. Denotamos por $vEB(u)$ uma árvore vEB com tamanho de universo de u e, a menos que u seja igual ao tamanho-base 2, o atributo *summary* aponta para uma árvore ($\sqrt[\downarrow]{u}$) e o arranjo *cluster* 0..$\sqrt[\uparrow]{u}$ −1 aponta para $\sqrt[\uparrow]{u}$ **árvores** *vEB* ($\sqrt[\downarrow]{u}$). Como a Figura 20.5 ilustra, uma árvore vEB contém dois atributos não encontrados em uma estrutura proto-vEB:

- *min* armazena o elemento mínimo na árvore vEB, e
- *max* armazena o elemento máximo na árvore vEB.

Além do mais, o elemento armazenado em *min* não aparece em nenhuma das árvores recursivas *vEB* ($\sqrt[\downarrow]{u}$) para a qual o arranjo *cluster* aponta. Portanto, os elementos armazenados em uma árvore *v*EB ($\sqrt[\downarrow]{u}$) são V.*min* mais todos os elementos armazenados recursivamente nas árvores *v*EB ($\sqrt[\downarrow]{u}$) para as quais o V.*cluster* 0.. $\sqrt[\downarrow]{u}$ 1 aponta. Observe que, quando uma árvore *vEB* contém dois ou mais elementos, tratamos *min* e *max* de modos diferentes: o elemento

armazenado em *min* não aparece em nenhum dos grupos, mas, exceto quando a árvore contém só um elemento (e nesse caso o máximo e o mínimo coincidem), o elemento armazenado em *max* aparece.

Visto que o tamanho-base é 2, uma árvore *vEB*(2) não precisa do arranjo A que a estrutura *proto-vEB*(2) correspondente tem. Em vez disso, podemos determinar seus elementos por seus atributos *min* e *max*. Em uma árvore *vEB* sem nenhum elemento, independentemente do tamanho do universo *u*, *min* e *max* são NIL.

A Figura 20.6 mostra uma árvore *vEB*(16) que mantém o conjunto {2, 3, 4, 5, 7, 14, 15}. Como o menor elemento é 2, V.*min* é igual a 2 e, apesar de o que high(2) = 0, o elemento 2 não aparece na árvore *vEB*(4) para a qual V.*cluster*[0] aponta: observe que V.*cluster*[0]: *min* é igual a 3 e, portanto, 2 não está nessa árvore *vEB*. De modo semelhante, visto que V.*cluster*[0] :*min* é igual a 3, e 2 e 3 são os únicos elementos em V.*cluster*[0], os grupos *vEB*(2) dentro de V.*cluster*[0] são vazios.

Os atributos *min* e *max* serão fundamentais para reduzir o número de chamadas recursivas dentro de operações em árvores *vEB*. Esses atributos nos ajudarão de quatro modos:

1. As operações MINIMUM e MAXIMUM nem mesmo precisam executar recursão porque podem apenas retornar os valores de *min* ou *max*.
2. A operação SUCCESSOR pode evitar fazer uma chamada recursiva para determinar se o sucessor de um valor x se encontra dentro de high(x). Isso porque o sucessor de x se encontra dentro de seu grupo se e somente se x é estritamente menor que o atributo *max de seu* grupo. Um argumento simétrico é válido para PREDECESSOR e *min*.
3. Podemos dizer se uma árvore vEB não tem nenhum elemento, exatamente um elemento ou no mínimo dois elementos em tempo constante por seus valores *min* e *max*. Essa capacidade nos ajudará nas operações INSERT e DELETE. Se *min* e *max* são NIL, então a árvore vEB não tem nenhum elemento. Se *min* e *max* são não NIL, mas um é igual ao outro, então a árvore vEB tem exatamente um elemento. Caso contrário, *min* e *max* são não NIL, mas não são iguais, e a árvore vEB tem dois ou mais elementos.
4. Se sabemos que uma árvore vEB é vazia, podemos inserir um elemento atualizando somente seus atributos *min* e *max*. Por consequência, podemos inserir em uma árvore vEB vazia em tempo constante. De modo semelhante, se sabemos que uma árvore vEB tem somente um elemento, podemos eliminar esse elemento em tempo constante atualizando somente *min* e *max*. Essas propriedades nos permitirão abreviar a cadeia de chamadas recursivas.

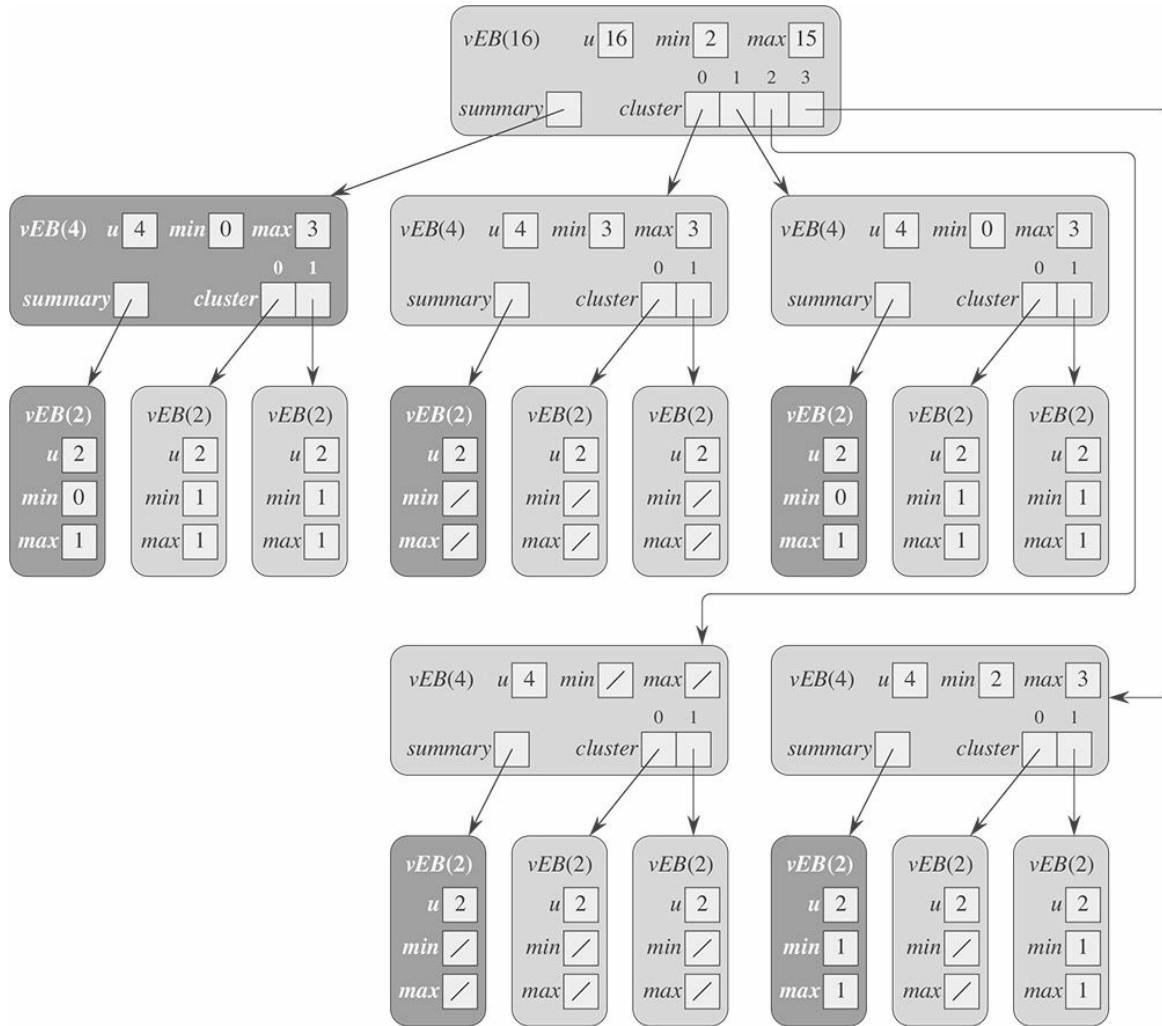
Ainda que o tamanho de universo *u* seja uma potência ímpar de 2, notamos que a diferença entre os tamanhos da árvore vEB resumo e dos grupos não afetará os tempos de execução assintóticos das operações de árvore vEB. Os tempos de execução de todos os procedimentos recursivos que implementam as operações de árvore vEB serão caracterizados pela recorrência

$$T(u) \le T\left(\sqrt[\uparrow]{u}\right) + O(1). \qquad (20.4)$$

Essa recorrência parece semelhante à recorrência (20.2), e nós a resolveremos de maneira semelhante. Fazendo $m = \lg u$, podemos reescrevê-la como

$$\mathrm{T}(2m) \le \mathrm{T}(^{\lceil \mathrm{d}m/2 \rceil}) + O(1).$$

Observando que $m/2 \le 2m/3$ para todo $m \ge 2$, temos

$$\mathrm{T}(2m) \le \mathrm{T}(^{2m/3}) + O(1).$$

Fazendo $S(m) = T(2m)$, reescrevemos esta última recorrência como

$$S(m) \le S(2m/3) + O(1)\ ,$$

cuja solução, pelo caso 2 do método mestre, é $S(m) = O(\lg m)$. (Em termos da solução assintótica, a fração 2/3 não faz nenhuma diferença em comparação com a fração 1/2 porque, quando aplicamos o método mestre, constatamos que $\log_{3/2} 1 = \log_2 1 = 0$.) Assim, temos $T(u) = T(2\ ) = S(m) = O\ (\lg m) = O(\lg \lg u)$.

Antes de usar uma árvore de van Emde Boas, temos de saber qual é o tamanho de universo u, para que possamos criar uma árvore de van Emde Boas do tamanho adequado que represente inicialmente um conjunto vazio. Como o Problema 20-1 pede que você mostre, o requisito de espaço total de uma árvore de van Emde Boas é $O(u)$, e criar uma árvore vazia no tempo $O(u)$ é uma operação óbvia. Por comparação, podemos criar uma árvore vermelho-preto vazia em tempo constante. Portanto, poderia não ser uma boa ideia usar uma árvore de van Emde Boas quando executamos somente um número pequeno de operações, visto que o tempo para criar a estrutura de dados seria maior que o tempo poupado nas operações individuais. Em geral, essa desvantagem não é significativa, já que, normalmente, usamos uma estrutura de dados simples, como um arranjo ou lista ligada, para representar um conjunto que tenha somente um pequeno número de elementos.

Figura 20.6 Uma árvore $vEB(16)$ correspondente à árvore proto-vEB da Figura 20.4. Ela armazena o conjunto {2, 3, 4, 5, 7, 14, 15}. Barras inclinadas indicam valores NIL. O valor armazenado no atributo *min* de uma árvore vEB não aparece em nenhum de seus grupos. O sombreado em tom mais escuro tem a mesma finalidade que o da Figura 20.4.

20.3.2 OPERAÇÕES EM UMA ÁRVORE DE VAN EMDE BOAS

Agora, estamos prontos para ver como executar operações em uma árvore de van Emde Boas. Como fizemos para a estrutura proto-van Emde Boas, consideraremos em primeiro lugar as operações de consulta e, em seguida, INSERT e DELETE. Devido à ligeira assimetria entre os elementos mínimo e máximo em uma árvore vEB — quando uma árvore vEB contém no mínimo dois elementos, o elemento mínimo não aparece dentro de um grupo, mas o elemento

máximo aparece — daremos pseudocódigo para todas as cinco operações de consulta. Como ocorre nas operações em estruturas proto-van Emde Boas, aqui as operações que adotam os parâmetros V e x, onde V é uma árvore de van Emde Boas e x é um elemento, supõem que $0 \le x < V.u$.

Encontrando os elementos mínimo e máximo

Como armazenamos o mínimo e o máximo nos atributos *min* e *max*, duas das operações têm somente uma linha e demoram tempo constante:

```
vEB-TREE -MINIMUM(V)
1 return V.min
```

```
vEB-TREE -MAXIMUM(V)
1 return V.max
```

Determinando se um valor está no conjunto

O procedimento vEB-TREE-MEMBER (V, x) tem um caso recursivo como o de PROTO-VEB-MEMBER, mas o caso-base é um pouco diferente. Verificamos também, diretamente se x é igual ao elemento mínimo ou ao elemento máximo. Visto que uma árvore vEB não armazena bits como uma estrutura proto-vEB faz, projetamos vEB-TREE−MEMBER para retornar TRUE ou FALSE em vez de 1 ou 0.

```
vEB-TREE-MEMBER (V, x)
1  if x == V.min ou x == V.max
2        return TRUE
3  elseif V.u == 2
4        return FALSE
5  else return vEB-TREE-MEMBER (V.cluster[high(x)], low(x))
```

A linha 1 verifica se x é igual ou ao elemento mínimo ou ao elemento máximo. Se for, a linha 2 retorna TRUE. Caso contrário, a linha 3 testa para o caso-base. Visto que uma árvore vEB(2)não tem nenhum elemento exceto os que estão em *min* e *max*, se for o caso-base, a linha 4 retorna FALSE. A outra possibilidade — não é um caso-base e x não é igual a *min* nem a *max* — é tratada pela chamada recursiva na linha 5.

A recorrência (20.4) caracteriza o tempo de execução do procedimento vEB-TREE−MEMBER e, portanto, esse procedimento leva o tempo $O(\lg \lg u)$.

Achando o sucessor e o predecessor

Em seguida, vemos como implementar uma operação SUCCESSOR. Lembre-se de que o procedimento PROTO-vEB-SUCCESSOR (V, x) podia fazer duas chamadas recursivas: uma para determinar se o sucessor de x reside no mesmo grupo que x e, se não residir, uma para encontrar o grupo que contém o sucessor de x. Como podemos acessar o valor máximo em uma árvore vEB rapidamente, podemos evitar de fazer duas chamadas recursivas, fazendo, em vez disso, uma chamada recursiva em um grupo ou no resumo, mas não em ambos.

```
vEB-TREE-SUCCESSOR(V, x)
1  if V.u == 2
2        if x == 0 e V.max == 1
3              return 1
4        else return NIL
5  elseif V.min ≠ NIL e x < V.min
6        return V:min
7  else max-low = vEB-TREE-MAXIMUM (V.cluster[high(x)])
8        if max-low ≠ NIL e low(x) < max-low
9              offset = vEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
10             return index(high(x), offset)
11       else succ-cluster = VEB-TREE-SUCCESSOR(V.summary, high(x))
12             if succ-cluster == NIL
13                  return NIL
14             else offset = vEB-TREE-MINIMUM(V.cluster[succ-cluster])
15                  return index(succ-cluster, offset)
```

Esse procedimento tem cinco instruções **return** e vários casos. Começamos com o caso-base nas linhas 2-4, que devolve 1 na linha 3 se estivermos tentando encontrar o sucessor de 0 e 1 estiver no conjunto de 2 elementos; caso contrário, o caso-base devolve NIL na linha 4.

Se não estivermos no caso-base, em seguida, verificamos na linha 5 se x é estritamente menor que o elemento mínimo. Se for, simplesmente devolvemos o elemento mínimo na linha 6.

Se chegarmos à linha 7, sabemos que não estamos em um caso-base e que x é maior ou igual ao valor mínimo na árvore vEB V. A linha 7 atribui a $max\text{-}low$ o elemento máximo no grupo de x. Se o grupo de x contém algum elemento maior que x, sabemos que o sucessor de x se encontra em algum lugar dentro do grupo de x. A linha 8 testa essa condição. Se o sucessor de x estiver dentro do grupo de x, a linha 9 determina em que lugar do grupo ele está, e a linha 10 retorna o sucessor do mesmo modo que a linha 7 de PROTO-vEB-SUCCESSOR.

Chegamos à linha 11, se x é maior ou igual ao maior elemento em seu grupo. Nesse caso, as linhas 11-15 encontram o sucessor de x do mesmo modo que as linhas 8-12 de PROTO-vEB-SUCCESSOR.

É fácil ver como a recorrência (20.4) caracteriza o tempo de execução de vEB-TREE-SUCCESSOR. Dependendo do resultado do teste na linha 7, o procedimento chama a si mesmo recursivamente na linha 9 (em uma árvore vEB com tamanho de universo $\sqrt[\downarrow]{u}$) ou na linha 11 (em uma árvore vEB com tamanho de universo $\sqrt[\uparrow]{u}$). Em qualquer dos casos, a única chamada recursiva é em uma árvore vEB com tamanho de universo no máximo $\sqrt[\uparrow]{u}$. O restante do procedimento, incluindo as chamadas a vEB-TREE-MINIMUM e vEB-TREE-MAXIMUM, leva o tempo $O(1)$. Por consequência, vEB-TREE −SUCCESSOR é executado no tempo do pior caso $O(\lg \lg u)$.

O procedimento vEB-TREE-PREDECESSOR é simétrico ao procedimento vEB-TREE-SUCCESSOR, porém com um caso adicional.

```
vEB-TREE-PREDECESSOR(V, x)
 1 if V.u == 2
 2     if x == 1 e V.min == 0
 3         return 0
 4     else return NIL
 5 elseif V.max ≠ NIL e x > V.max
 6     return V.max
 7 else min-low = vEB-TREE-MINIMUM(V.cluster[high(x)])
 8     if min-low ≠ NIL e low(x)> min-low
 9         offset = vEB-TREE-PREDECESSOR(V.cluster[high(x)], low(x))
10         return index(high(x), offset)
11     else pred-cluster = vEB-TREE-PREDECESSOR(V.summary, high(x))
12         if pred-cluster == NIL
13             if V.min ≠ NIL e x > V.min
14                 return V.min
15             else return NIL
16         else offset = vEB-TREE-MAXIMUM(V.cluster[pred-cluster])
17             return index(pred-cluster, offset)
```

As linhas 13-14 formam o caso adicional. Esse caso ocorre quando o predecessor de x, se existir, não reside no grupo de x. Em vEB-TREE-SUCCESSOR, tínhamos certeza de que, se o sucessor de x reside fora do grupo de x, deve residir em um grupo de número mais alto. Porém, se o predecessor de x é o valor mínimo na árvore vEB V, então, definitivamente, o predecessor não reside em nenhum grupo. A linha 13 verifica essa condição e a linha 14 devolve o valor mínimo como adequado.

Esse caso extra não afeta o tempo de execução assintótico de vEB-TREE-PREDECESSOR quando comparado com vEB-TREE-SUCCESSOR e, portanto, vEB-TREE-PREDECESSOR é executado no tempo do pior caso $O(\lg \lg u)$.

Inserindo um elemento

Agora, examinamos como inserir um elemento em uma árvore vEB. Lembre-se de que PROTO-vEB INSERT fazia duas chamadas recursivas: uma para inserir o elemento e uma para inserir o número do grupo do elemento no resumo. O procedimento vEB-TREE-INSERT fará somente uma chamada recursiva. Como podemos nos safar com apenas uma? Quando inserimos um elemento, o grupo no qual ele entra já tem um outro elemento ou não tem. Se o grupo já tiver um outro elemento, o número do grupo já está no resumo e, assim, não precisamos fazer aquela chamada recursiva. Se o grupo ainda não tiver nenhum outro elemento, o elemento que está sendo inserido torna-se o único elemento no grupo e não precisamos executar recursão para inserir um elemento em uma árvore vEB vazia:

```
vEB-EMPTY-TREE-INSERT(V, x)
1 V.min = x
2 V.max = x
```

Com esse procedimento em mãos, apresentamos aqui o pseudocódigo para vEB-TREE-INSERT(V, x), que supõe que x ainda não é um elemento no conjunto representado pela árvore vEB V:

```
vEB-TREE-INSERT.V(x)
 1 if V.min == NIL
 2     vEB-EMPTY-TREE-INSERT(V, x)
 3 else if x < V.min
 4         troque x com V.min
 5     if V.u > 2
 6         if vEB-TREE-MINIMUM(V.cluster[high(x)] == NIL
 7             vEB-TREE-INSERT(V.summary, high(x))
 8             vEB-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
 9         else vEB-TREE-INSERT(V.cluster[high(x)], low(x))
10     if x > V.max
11         V.max = x
```

Esse procedimento funciona da seguinte maneira: a linha 1 testa se V é uma árvore vEB vazia e, se for, a linha 2 trata esse caso fácil. As linhas 3-11 sabem que V não é vazia e, portanto, algum elemento será inserido em um dos grupos de V. Mas esse elemento poderia não ser necessariamente o elemento x passado para vEB-Tree-Insert. Se $x < min$, como testado na linha 3, então x precisa tornar-se o novo min. Todavia, não queremos perder o min original e, por isso, precisamos inseri-lo em um dos grupos de V. Nesse caso, a linha 4 permuta x por min, de modo que inserimos o min original em um dos grupos de V.

Executamos as linhas 6-9 somente se V não for uma árvore vEB do caso-base. A linha 6 determina se o grupo para o qual x irá está atualmente vazio. Se estiver, a linha 7 insere o número do grupo de x no resumo e a linha 8 trata o caso fácil, que é inserir x em um grupo vazio. Se o grupo de x não estiver vazio no momento em questão, a linha 9 insere x em seu grupo. Nesse caso, não precisamos atualizar o resumo, visto que o número do grupo de x já é um membro do resumo.

Finalmente, as linhas 10-11 cuidam de atualizar max se $x > max$. Observe que, se V é uma árvore vEB do caso-base, as linhas 3-4 e 10-11 atualizam min e max *adequadamente.*

Mais uma vez, é fácil ver como a recorrência (20.4) caracteriza o tempo de execução. Dependendo do resultado do teste na linha 6, é executada a chamada recursiva na linha 7 (em uma vEB com tamanho de universo $\sqrt[\uparrow]{u}$) ou a chamada recursiva na linha 9 (em uma vEB com tamanho de universo $\sqrt[\downarrow]{u}$). Em qualquer dos casos, essa única chamada recursiva é numa árvore vEB com tamanho de universo no máximo $\sqrt[\uparrow]{u}$. Como o restante de vEB-Tree-Insert leva o tempo O(1), a recorrência (20.4) se aplica e, assim, o tempo de execução é O(lg lg u).

Eliminando um elemento

Finalmente, examinamos como eliminar um elemento de uma árvore vEB. O procedimento vEB-Tree-Delete.V(x) supõe que x é, no momento considerado, um elemento no conjunto representado pela árvore vEB V.

```
vEB-TREE-DELETE(V, x)
1  if V.min == V.max
2    V.min = NIL
3    V.max = NIL
4  elseif V.u == 2
5    if x == 0
6        V.min = 1
7    else V.min = 0
8    V.max = V.min
9  else if x == V.min
10           first-cluster = vEB-TREE-MINIMUM.(V.summary)
11           x = index.first-cluster;
               vEB-TREE-MINIMUM.V.cluster[first-cluster]))
12           V.min = x
13       vEB-TREE-DELETE.(V.cluster[high(x)] low(x))
14       if vEB-TREE-MINIMUM.(V.cluster[high(x)] == NIL
15           vEB-TREE-DELETE.V.summary; high(x))
16           if x == V.max
17               summary-max = vEB-TREE-MAXIMUM.(V.summary)
18               if summary-max == NIL
19                   V.max = V.min
20               else V.max = index.summary-max;
                         vEB-TREE-MAXIMUM.(V.cluster[summary-max]))
21       elseif x == V.max
22           V.max = index.high(x),
               vEB-TREE-MAXIMUM.V.cluster[high(x)]))
```

O procedimento vEB-TREE-DELETE funciona da maneira descrita a seguir. Se a árvore vEB V contém somente um elemento, eliminá-lo é tão fácil quanto inseri-lo em uma árvore vEB vazia: basta definir *min* e *max* como NIL. As linhas 1-3 tratam desse caso. Do contrário, V tem, no mínimo, dois elementos. A linha 4 testa se V é um caso-base de árvore vEB e, se for, as linhas 5-8 definem *min* e *max* como o único elemento remanescente.

As linhas 9-22 levam em conta que V tem dois ou mais elementos e que $u \geq 4$. Nesse caso, teremos de eliminar um elemento de um grupo. Contudo, o elemento que eliminamos de um grupo poderia não ser x porque, se x é igual a *min*, uma vez eliminado x algum outro elemento dentro de um dos grupos de V torna-se o novo *min* e temos de eliminar o outro elemento desse grupo. Se o teste na linha 9 revelar que estamos nesse caso, a linha 10 define *first-cluster* como o número do grupo que contém o elemento mais baixo exceto *min*, e a linha 11 define x como o valor do elemento mais baixo naquele grupo. Esse elemento torna-se o novo *min* na linha 12 e, já que definimos x como esse valor, ele é o elemento que eliminaremos desse grupo.

Quando alcançamos a linha 13, sabemos que precisamos eliminar o elemento x de seu grupo, quer x fosse o valor originalmente passado para vEB-TREE-DELETE, quer x seja o elemento que está se tornando o novo mínimo. A linha 13 elimina x desse grupo. Agora, aquele grupo poderia tornar-se vazio, o que a linha 14 testa; se ele se tornou vazio, precisamos remover o número de grupo de x do resumo, o que a linha 15 faz. Após atualizar o resumo, poderia ser necessário atualizar *max*. A linha 16 verifica se estamos eliminando o elemento máximo em V e, se estivermos, a linha 17 define *summary-max* como o número do grupo não vazio de número mais alto. (A chamada vEB-TREE-MAXIMUM (*V.summary*) funciona porque já chamamos vEB-TREE-DELETE recursivamente em *V.summary* e, portanto, *V.summary.max* já foi atualizado conforme necessário.) Se todos os grupos de V são vazios, o único elemento restante em V é *min*; a linha 18 verifica esse caso, e a linha 19 atualiza *max* adequadamente. Caso contrário, a linha 20 define *max* como o elemento máximo no grupo não vazio de número mais alto. (Se é desse grupo que o elemento foi eliminado, novamente confiamos que a chamada recursiva na linha 13 já tenha corrigido o atributo *max* daquele grupo.)

Finalmente, temos de tratar o caso no qual o grupo de x não se tornou vazio em razão da eliminação de x. Embora não tenhamos de atualizar o resumo nesse caso, poderíamos ter de atualizar *max*. A linha 21 testa esse caso e, se tivermos de atualizar *max*, a linha 22 o faz (mais uma vez confiando que a chamada recursiva tenha corrigido *max no* grupo).

Agora mostramos que vEB-TREE-DELETE é executada no tempo O(lg lg u) do pior caso. À primeira vista, você poderia imaginar que a recorrência (20.4) nem sempre se aplica porque uma única chamada de vEB-TREE-DELETE pode fazer duas chamadas recursivas: uma na linha 13 e uma na linha 15. Embora o procedimento possa fazer ambas as chamadas recursivas, vamos pensar no que acontece quando ele o faz. Para que a chamada recursiva na linha 15 ocorra, o teste na linha 14 deve mostrar que o grupo de x está vazio. O único modo possível de o grupo de x estar vazio é se x era o único elemento nesse grupo quando fizemos a chamada recursiva na linha 13. Porém, se x era o único elemento nesse grupo, aquela chamada recursiva levou o tempo $O(1)$ porque só as linhas 1-3 foram executadas. Assim, temos duas possibilidades mutuamente exclusivas:

- A chamada recursiva na linha 13 demorou tempo constante.
- A chamada recursiva na linha 15 não ocorreu.

Em qualquer caso, a recorrência (20.4) caracteriza o tempo de execução de vEB-TREE-DELETE e, por conseqûencia, seu tempo de execução do pior caso é $O(\lg \lg u)$.

Exercícios

20.3-1 Modifique árvores vEB para suportar chaves duplicadas.

20.3-2 Modifique árvores vEB para suportar chaves que tenham dados satélites associados.

20.3-3 Escreva pseudocódigo para um procedimento que cria uma árvore de van Emde Boas vazia.

20.3-4 O que acontece se você chamar vEB-TREE-INSERT com um elemento que já está na árvore vEB? O que acontece se você chamar vEB-TREE-DELETE com um elemento que não está na árvore vEB? Explique por que os procedimentos exibem o comportamento que exibem. Mostre como modificar árvores vEB e suas operações de modo que possamos verificar, em tempo constante, se um elemento está presente.

20.3-5 Suponha que, em vez de $\sqrt[\uparrow]{u}$ grupos, cada um com tamanho de universo $\sqrt[\downarrow]{u}$, construíssemos árvores vEB que tivessem $u_{1/k}$ grupos, cada um com tamanho de universo $u^{1-1/k}$, onde $k > 1$ é uma constante. Se tivéssemos de modificar as operações adequadamente, quais seriam seus tempos de execução? Para a finalidade de análise, considere que $u_1/^k$ e $u_{1-1}/^k$ são sempre inteiros.

20.3-6 Criar uma árvore vEB com tamanho de universo u requer tempo $O(u)$ Suponha que desejamos dar conta explicitamente desse tempo. Qual é o menor número de operações n para o qual o tempo amortizado de cada operação em uma árvore vEB é $O(\lg \lg u)$?

Problemas

20-1 Requisitos de espaço para árvores de van Emde Boas

Esse problema explora os requisitos de espaço para árvores de van Emde Boas e sugere um modo de modificar a estrutura de dados para que esse requisito de espaço dependa do número n de elementos que estão realmente armazenados na árvore, em vez de depender do tamanho de universo u. Por simplicidade, considere que $\sqrt{u}$ seja sempre um inteiro.

a. Explique por que a seguinte recorrência caracteriza o requisito de espaço $P(u)$ de uma árvore de van Emde Boas com tamanho de universo u:

$$P(u) = \left(\sqrt{u}+1\right)P\left(\sqrt{u}\right)+\Theta\left(\sqrt{u}\right) \tag{20.5}$$

b. Prove que a recorrência (20.5) tem a solução $P(u) = O(u)$.

Para reduzir os requisitos de espaço, vamos definir uma ***árvore de van Emde Boas de espaço reduzido***, ***ou árvore vEB-ER***, como uma árvore vEB V, porém com as seguintes mudanças:

- Em vez de o atributo V.*cluster* ser armazenado como um simples arranjo de ponteiros para árvores vEB com tamanho de universo $\sqrt{u}$, ele é uma tabela de espalhamento (veja o Capítulo 11) armazenada como uma tabela dinâmica (veja a Seção 17.4). Assim como a versão de arranjo de V.*cluster*, a tabela de espalhamento armazena ponteiros para árvores vEB-ER com tamanho de universo $\sqrt{u}$. Para encontrar o *i*--ésimo grupo, consultamos a chave *i* na tabela de espalhamento, de modo a podermos encontrar o *i*-ésimo grupo com uma única busca na tabela de espalhamento.

- A tabela de espalhamento armazena somente ponteiros para grupos não vazios. Uma busca na tabela de espalhamento por um grupo vazio retorna NIL, o que indica que o grupo é vazio.

- O atributo V.*summary* é NIL se todos os grupos são vazios. Caso contrário, V.*summary* aponta para uma árvore vEB-ER com tamanho de universo $\sqrt{u}$.

Como a tabela de espalhamento é implementada com uma tabela dinâmica, o espaço que ela requer é proporcional ao número de grupos não vazios.

Quando precisamos inserir um elemento em uma árvore vEB-ER vazia, criamos a árvore vEB-RS chamando o seguinte procedimento, onde o parâmetro u é o tamanho de universo da árvore vEB-RS:

```
CREATE-NEW-ER-vEB-TREE(u)
1  alocar uma nova árvore vEB V
2  V.u = u
3  V.min = NIL
4  V.max = NIL
5  V.summary = NIL
6  criar V.cluster como uma tabela de espalhamento dinâmica vazia
7  return V
```

c. Modifique o procedimento vEB-TREE-INSERT para produzir pseudocódigo para o procedimento ER-vEB-TREE-INSERT.(V, x), que insere x na árvore vEB-ER V, chamando CREATE-NEW-ER-vEB-TREE adequadamente.

d. Modifique o procedimento vEB-TREE-SUCCESSOR para produzir pseudocódigo para o procedimento ER-vEB-TREE-SUCCESSOR$V(x)$, que devolve o sucessor de x na árvore vEB-ER V ou NIL se x não tiver nenhum sucessor em V.

e. Prove que, sob a premissa de hashing uniforme simples, seus procedimentos RS-vEB-TREE-INSERT e ER-vEB-TREE-SUCCESSOR são executados no tempo amortizado esperado $O(\lg \lg u)$.

f. Supondo que elementos de uma árvore vEB nunca são eliminados, prove que o requisito de espaço para a estrutura árvore vEB-ER é $O(n)$, onde n é o número de elementos realmente armazenados na árvore vEB-ER.

g. Árvores vEB-ER têm uma outra vantagem em reação a árvores vEB: criá-las exige menos tempo. Quanto tempo demora para criar uma árvore vEB-ER vazia?

20-2 *y-fast tries*

Esse problema investiga as "***y***-fast tries" ("tries y rápidas", "árvores digitais y rápidas") de D. Willard que, como as árvores de van Emde Boas, executam cada uma das operações MEMBER, MINIMUM, MAXIMUM, PREDECESSOR e SUCCESSOR em elementos extraídos de um universo com tamanho u no tempo do pior caso $O(\lg \lg u)$. As operações INSERT e DELETE demoram o tempo amortizado $O(\lg \lg u)$. Do mesmo modo que as árvores de van Emde Boas de espaço reduzido (veja o Problema 20-1), as y-fast tries usam somente o espaço $O(n)$ para armazenar n elementos. O projeto de y-fast tries depende de hashing perfeito (veja a Seção 11.5).

Como um estrutura preliminar, suponha que criamos uma tabela de hash perfeito que contenha não somente todo elemento no conjunto dinâmico, mas também todo prefixo da representação binária de todo elemento no conjunto. Por exemplo, se $u = 16$, de modo que $\lg u = 4$, e $x = 13$ está no conjunto, então, como a representação binária de 13 é 1101, a tabela de hash perfeito contém as cadeias 1, 11, 110 e 1101. Além da tabela de espalhamento, criamos uma lista duplamente ligada dos elementos presentes atualmente no conjunto, em ordem crescente.

a. Quanto espaço essa estrutura requer?

b. Mostre como executar as operações MINIMUM e MAXIMUM no tempo $O(1)$; as operações MEMBER, PREDECESSOR e SUCCESSOR no tempo $O(\lg \lg u)$ e as operações INSERT e DELETE no tempo $O(\lg u)$.

Para reduzir o requisito de espaço para $O(n)$, fazemos as seguintes mudanças na estrutura de dados:

- Agrupamos os n elementos em $n/\lg u$ grupos de tamanho $\lg u$. (Suponha, por enquanto, que $\lg u$ seja um divisor de n.) O primeiro grupo consiste nos $\lg u$ menores elementos no conjunto, o segundo grupo consiste nos $\lg u$ menores elementos seguintes, e assim por diante.

- Designamos um valor "representante" para cada grupo. O representante do i-ésimo grupo é no mínimo tão grande quanto o maior elemento no i-ésimo grupo, e é menor que todo elemento do $(i + 1)$-ésimo grupo. (O representante do último grupo pode ser o máximo elemento possível u - 1.) Observe que um representante pode ser um valor que não está atualmente no conjunto.

- Armazenamos os $\lg u$ elementos de cada grupo em uma árvore de busca binária balanceada, tal como uma árvore vermelho-preto. Cada representante aponta para a árvore de busca binária balanceada para seu grupo, e cada árvore de busca binária balanceada aponta para seu representante de grupo.

- A tabela de espalhamento perfeito armazena somente os representantes, que são também armazenados em uma lista duplamente ligada em ordem crescente.

  Denominamos essa estrutura ***y-fast trie***.

c. Mostre que uma y-fast trie requer somente espaço $O(n)$ para armazenar n elementos.

d. Mostre como executar as operações MINIMUM e MAXIMUM no tempo $O(\lg \lg u)$ com uma y-fast trie.

e. Mostre como executar a operação MEMBER no tempo $O(\lg \lg u)$.

f. Mostre como executar as operações PREDECESSOR e SUCCESSOR no tempo $O(\lg \lg u)$.

g. Explique por que as operações INSERT e DELETE demoram o tempo $(\lg \lg u)$.

h. Mostre como relaxar o requisito de que cada grupo em uma y-fast trie deve ter exatamente $\lg u$ elementos para permitir que INSERT e DELETE sejam executadas no tempo amortizado $O(\lg \lg u)$ sem afetar os tempos de execução assintóticos das outras operações.

NOTAS DO CAPÍTULO

A estrutura de dados neste capítulo deve seu nome a P. van Emde Boas, que descreveu uma primeira forma da ideia em 1975 [339]. Artigos publicados mais tarde por van Emde Boas, Kaas e Zijlstra [341] refinaram a ideia e a exposição. Subsequentemente, Mehlhorn e Näher [252] ampliaram as ideias para aplicá-las a tamanhos de universo que são primos. O livro de Mehlhorn [249] contém um tratamento ligeiramente diferente de árvores de van Emde Boas que o utilizado neste capítulo.

Usando as ideias que fundamentam as árvores de van Emde Boas, Dementiev et al. [83] desenvolveram uma árvore de busca não recursiva de três níveis que rodou mais rapidamente do que as árvores de van Emde Boas em seus próprios experimentos.

Wang e Lin [347] projetaram uma versão das árvores de van Emde Boas com pipeline em hardware que consegue tempo amortizado constante por operação e usa $O(\lg \lg u)$ estágios no pipeline.

Um limite inferior determinado por Paˇ trasçu e Thorup [273, 274] para encontrar o predecessor mostra que árvores de van Emde Boas são ótimas para essa operação, mesmo que a aleatorização seja permitida.

---

[1] O Capítulo 13 não discute explicitamente como implementar extract-min e decrease-key, mas podemos construir facilmente essas operações para qualquer estrutura de dados que suporte minimum, delete e insert.

[2] Em todo este capítulo consideramos que Minimum e Maximum retornam nil se o conjunto dinâmico estiver vazio e que Successor e Predecessor retornam nil se o elemento que lhes é dado não tiver nenhum sucessor ou predecessor, respectivamente.

21 ESTRUTURAS DE DADOS PARA CONJUNTOS DISJUNTOS

Algumas aplicações envolvem agrupar n elementos distintos em uma coleção de conjuntos disjuntos. Muitas vezes, essas aplicações precisam executar duas operações em particular: encontrar o único conjunto que contém um dado elemento e unir dois conjuntos. Este capítulo explora métodos para manter uma estrutura de dados que suporta essas operações.

A Seção 21.1 descreve as operações suportadas por uma estrutura de dados de conjuntos disjuntos e apresenta uma aplicação simples. Na Seção 21.2, examinaremos uma implementação de lista ligada simples para conjuntos disjuntos. A Seção 21.3. apresenta uma representação mais eficiente que utiliza árvores enraizadas. O tempo de execução com a utilização da representação de árvore é teoricamente superlinear, porém, para todas as finalidades práticas, é linear. A Seção 21.4 define e discute uma função de crescimento muito rápido e sua inversa de crescimento muito lento, que aparece no tempo de execução de operações na implementação baseada em árvore e, em seguida, por uma análise amortizada complexa, prova um limite superior para o tempo de execução que mal é superlinear.

21.1 OPERAÇÕES EM CONJUNTOS DISJUNTOS

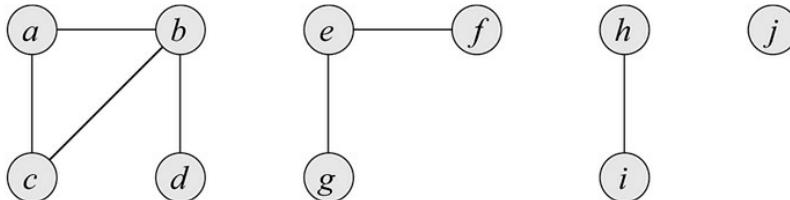
Uma ***estrutura de dados de conjuntos disjuntos*** mantém uma coleção $S = \{S_1, S_2, ..., S_k\}$ de conjuntos dinâmicos disjuntos. Identificamos cada conjunto por um ***representante***, que é algum membro do conjunto. Em algumas aplicações, não importa qual membro seja usado como representante; a única coisa que importa é que, se solicitarmos o representante de um conjunto dinâmico duas vezes sem modificar o conjunto entre as solicitações, obteremos a mesma resposta ambas as vezes. Outras aplicações podem exigir uma regra previamente especificada para escolher o representante, como escolher o menor elemento no conjunto (considerando, é claro, que os elementos podem ser ordenados).

Como nas outras implementações de conjuntos dinâmicos que estudamos, representamos cada elemento de um conjunto por um objeto. Denotando a representação de um objeto por x, desejamos suportar as seguintes operações:

MAKE-SET(x) cria um novo conjunto cujo único membro (e, portanto, o representante) é x. Visto que os conjuntos são disjuntos, exigimos que x ainda não esteja em algum outro conjunto.

UNION(x, y) une os conjuntos dinâmicos que contêm x e y, digamos S_x e S_y, em um novo conjunto que é a união desses dois conjuntos. Supomos que os dois conjuntos são disjuntos antes da operação. O representante do conjunto resultante é qualquer membro de $S_x \cup S_y$, embora muitas implementações de UNION escolham especificamente o representante de S_x ou o de S_y como o novo representante. Visto que exigimos que os conjuntos na coleção sejam disjuntos, conceitualmente "destruímos" os conjuntos S_x e S_y, removendo-os da coleção S. Na prática, frequentemente absorvemos os elementos de um dos conjuntos no outro conjunto.

FIND-SET(x) retorna um ponteiro para o representante do (único) conjunto que contém x.

Ao longo deste capítulo, analisaremos os tempos de execução de estruturas de dados de conjuntos disjuntos em termos de dois parâmetros: n, o número de operações Make-Set, e m, o número total de operações Make-Set, Union e Find-Set. Visto que os conjuntos são disjuntos, cada operação Union reduz o número de conjuntos de uma unidade. Portanto, após n - 1 operações Union, resta apenas um conjunto. Assim, o número de operações Union é no máximo n - 1. Observe também que, como as operações Make-Set estão incluídas no número total de operações m, temos $m \geq n$. Supomos que as n operações Make-Set são as primeiras n operações executadas.

Uma aplicação de estruturas de dados de conjuntos disjuntos

Uma das muitas aplicações de estruturas de dados de conjuntos disjuntos surge na determinação das componentes conexas de um grafo não dirigido (veja a Seção B.4). Por exemplo, a Figura 21.1(a) mostra um grafo com quatro componentes conexas.

O procedimento Connected-Components que apresentamos a seguir, utiliza as operações de conjuntos disjuntos para calcular as componentes conexas de um grafo. Tão logo Connected- Components tenha pré-processado o grafo, o procedimento Same-Component responde, por consulta, se dois vértices estão na mesma componente conexa.[1] (No pseudocódigo, denotamos o conjunto de vértices de um gráfico G por G.V e o conjunto de vértices por G.E.)

```
Connected-Components(G)
1    for cada vértice v ∈ G.V
2        Make-Set(v)
3    for cada aresta(u, v) ∈ G.E
4        if Find-Set(u) ≠ Find-Set(v)
5            Union(u, v)
```

(a)

Aresta processada	Coleção de conjuntos disjuntos								
conjuntos iniciais	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(b)

Figura 21.1 (**a**) Um grafo com quatro componentes conexas: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$ e $\{j\}$. (**b**) A coleção de conjuntos disjuntos após o processamento de cada aresta.

```
SAME-COMPONENT(u, v)
1   if FIND-SET(u) == FIND-SET(v)
2       return TRUE
3   else return FALSE
```

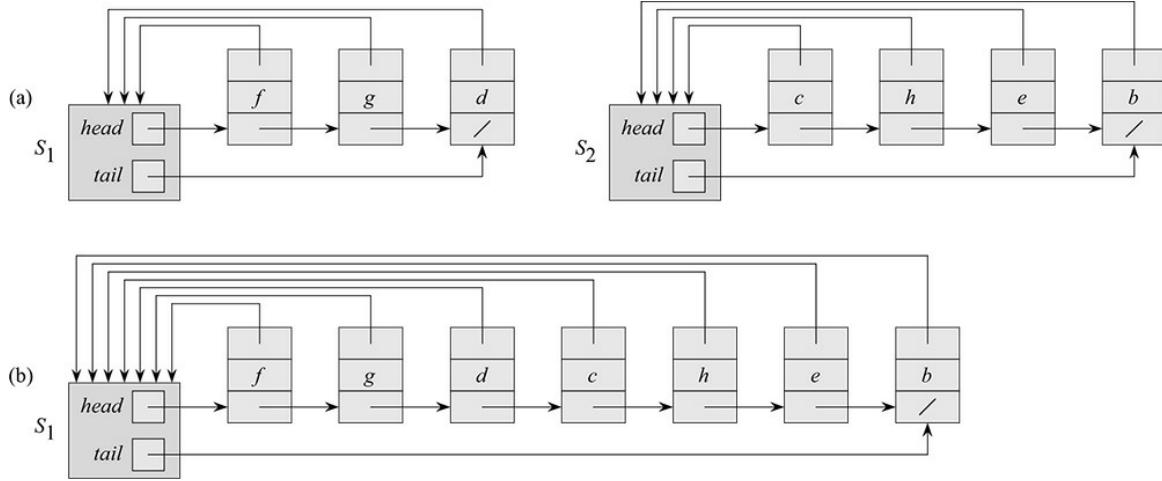
O procedimento CONNECTED-COMPONENTS inicialmente coloca cada vértice v em seu próprio conjunto. Em seguida, para cada aresta (u, v), ele une os conjuntos que contêm u e v. Pelo Exercício 21.1-2, após o processamento de todas as arestas, dois vértices estão na mesma componente conexa se e somente se os objetos correspondentes estão no mesmo conjunto. Assim, CONNECTED-COMPONENTS calcula conjuntos de um modo tal que o procedimento SAME-COMPONENT pode determinar se dois vértices estão na mesma componente conexa. A Figura 21.1(b) ilustra como os conjuntos disjuntos são calculados por CONNECTED-COMPONENTS.

Em uma implementação real desse algoritmo de componentes conexas, as representações do grafo e da estrutura de dados de conjuntos disjuntos precisariam referenciar uma à outra. Isto é, um objeto que representa um vértice conteria um ponteiro para o objeto conjunto disjunto correspondente e vice-versa. Esses detalhes de programação dependem da linguagem de implementação, e não os examinaremos mais aqui.

Exercícios

21.1-1 Suponha que CONNECTED-COMPONENTS seja executado em um grafo não dirigido $G = (V, E)$, onde $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ e as arestas de E sejam processadas na ordem (d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e). Faça uma lista de vértices em cada componente conexa após cada iteração das linhas 3-5.

21.1-2 Mostre que, depois de todas as arestas serem processadas por CONNECTED-COMPONENTS, dois vértices estão na mesma componente conexa se e somente se estão no mesmo conjunto.

21.1-3 Durante a execução de CONNECTED-COMPONENTS em um grafo não dirigido $G = (V, E)$ com k componentes conexas, quantas vezes FIND-SET é chamado? Quantas vezes UNION é chamado? Expresse suas respostas em termos de $|V|$, $|E|$ e k.

21.2 REPRESENTAÇÃO DE CONJUNTOS DISJUNTOS POR LISTAS LIGADAS

A Figura 21.2(a) mostra um modo simples de implementar uma estrutura de dados de conjuntos disjuntos: cada conjunto é representado por sua própria lista ligada. O objeto para cada conjunto tem atributos *head* (início), que aponta para o primeiro objeto na lista, e *tail* (fim), que aponta para o último objeto. Cada objeto na lista ligada contém um membro de conjunto, um ponteiro para o próximo objeto na lista e um ponteiro de volta para o objeto conjunto. Dentro de cada lista ligada, os objetos podem aparecer em qualquer ordem. O representante é o membro do conjunto no primeiro objeto na lista.

Com essa representação de lista ligada, MAKE-SET e FIND-SET são fáceis e exigem o tempo $O(1)$. Para executar MAKE-SET(x), criamos uma nova lista ligada cujo único objeto é x. Para FIND- -SET(x), simplesmente seguimos o ponteiro de x de volta ao seu objeto conjunto e depois retornamos o membro no objeto para o qual *head* aponta. Por exemplo, na Figura 21.2(a), a chamada FIND-SET(g) retornaria f.

Uma implementação simples de união

A implementação mais simples da operação UNION usando a representação de conjunto de lista ligada demora um tempo significativamente maior que MAKE-SET ou FIND-SET. Como mostra a Figura 21.2(b), executamos UNION(x, y) anexando a lista de y ao final da lista de x. Usamos o ponteiro *tail* para a lista de x para encontrar rapidamente onde anexar a lista de y. Como todos os membros da lista de y juntam-se à lista de x, podemos destruir o objeto conjunto para a lista de y.

Figura 21.2 (**a**) Representações de dois conjuntos por listas ligadas. O conjunto S_1 contém os membros d, f e g, com representante f, e o conjunto S_2 contém os membros b, c, e e h, com representante c. Cada objeto na lista contém um membro de conjunto, um ponteiro para o próximo objeto na lista e um ponteiro de volta ao objeto conjunto. Cada objeto conjunto tem ponteiros *head* (*início*) e *tail* (*final*) para o primeiro e o último objeto, respectivamente. (**b**) Resultado de UNION(g, e), que anexa a lista ligada que contém e a lista ligada que contém g. O representante do conjunto resultante é f. O objeto conjunto para a lista de e, S_2 , é destruído.

Infelizmente, temos de atualizar o ponteiro para o objeto conjunto para cada objeto que estava originalmente na lista de y, o que demora tempo linear em relação ao comprimento da lista de y. Na Figura 21.2, por exemplo, a operação UNION(g, e) provoca a atualização dos ponteiros nos objetos para b, c, e e h.

De fato, podemos construir facilmente uma sequência de m operações com *n objetos* que exige o tempo Q(n_2). Suponha que tenhamos objetos x_1, x_2, ..., x_n. Executamos a sequência de n operações MAKE-SET seguida por n - 1 operações UNION mostradas na Figura 21.3, de modo que m = 2n - 1. Gastamos o tempo Q(n) executando as n operações MAKE-SET. Como a i-ésima operação UNION atualiza i objetos, o número total de objetos atualizados por todas as n - 1 operações UNION é

$$\sum_{i=1}^{n-1} i = \Theta(n^2).$$

O número total de operações é 2n - 1 e, portanto, cada operação exige em média o tempo Q(n). Isto é, o tempo amortizado de uma operação é Q(n).

Uma heurística de união ponderada

No pior caso, tal implementação do procedimento UNION exige um tempo médio Q(n) por chamada porque é possível que estejamos anexando uma lista mais longa a uma lista mais curta; temos de atualizar o ponteiro para o objeto conjunto para cada membro da lista mais longa.

Operação	Número de objetos atualizados
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n - 1$

Figura 21.3 Uma sequência de $2n$ - 1 operações em n objetos que demora o tempo Q(n_2) ou o tempo Q(n) por operação, em média, usando a representação de conjuntos por listas ligadas e a implementação simples de UNION.

Em vez disso, suponha que cada lista também inclua o comprimento da lista (que podemos manter facilmente) e que sempre anexamos a lista mais curta à mais longa, rompendo ligações arbitrariamente. Com essa ***heurística de união ponderada*** simples, uma única operação UNION ainda pode demorar o tempo (n) se ambos os conjuntos têm (n) membros. Porém, como mostra o teorema a seguir, uma sequência de m operações MAKE-SET, UNION e FIND-SET, n das quais são operações MAKE-SET, demora o tempo $O(m + n \lg n)$.

Teorema 21.1

Usando a representação de lista ligada de conjuntos disjuntos e a heurística de união ponderada, uma sequência de m operações MAKE-SET, UNION e FIND-SET, n das quais são operações MAKE-SET, demora o tempo $O(m + n \lg n)$.

Prova Como cada operação UNION une dois conjuntos disjuntos, executamos no máximo n - 1 operações UNION no total. Agora, limitamos o tempo total gasto por essas operações UNION. Começamos determinando, para cada objeto, um limite superior para o número de vezes que o ponteiro do objeto de volta ao seu objeto conjunto é atualizado. Considere um determinado objeto x. Sabemos que cada vez que o ponteiro de x foi atualizado, x deve ter começado no conjunto menor. Portanto, na primeira vez que o ponteiro de x foi atualizado, o conjunto resultante devia conter no mínimo dois membros. De modo semelhante, na próxima vez que o ponteiro de x foi atualizado, o conjunto resultante devia ter no mínimo quatro membros. Continuando, observamos que, para qualquer $k \leq n$, depois que o ponteiro de x foi atualizado $\lg k$ vezes, o conjunto resultante deve ter contido no mínimo k membros. Visto que o conjunto maior tem no máximo n membros, o ponteiro de cada objeto é atualizado no máximo $\lg n$ vezes em todas as operações UNION. Assim, o tempo total gasto na atualização de ponteiros de objeto em todas as operações UNION é $O(n \lg n)$. Também

devemos dar conta da atualização dos ponteiros *tail* e dos comprimentos de listas, que demoram apenas o tempo Q(1) por operação UNION. Assim, o tempo total gasto em todas as operações UNION é $O(n \lg n)$.
O tempo para a sequência inteira de m operações decorre facilmente. Cada operação MAKE-SET e FIND-SET demora o tempo $O(1)$, e existem $O(m)$ dessas operações. Portanto, o tempo total para a sequência inteira é $O(m + n \lg n)$.

Exercícios

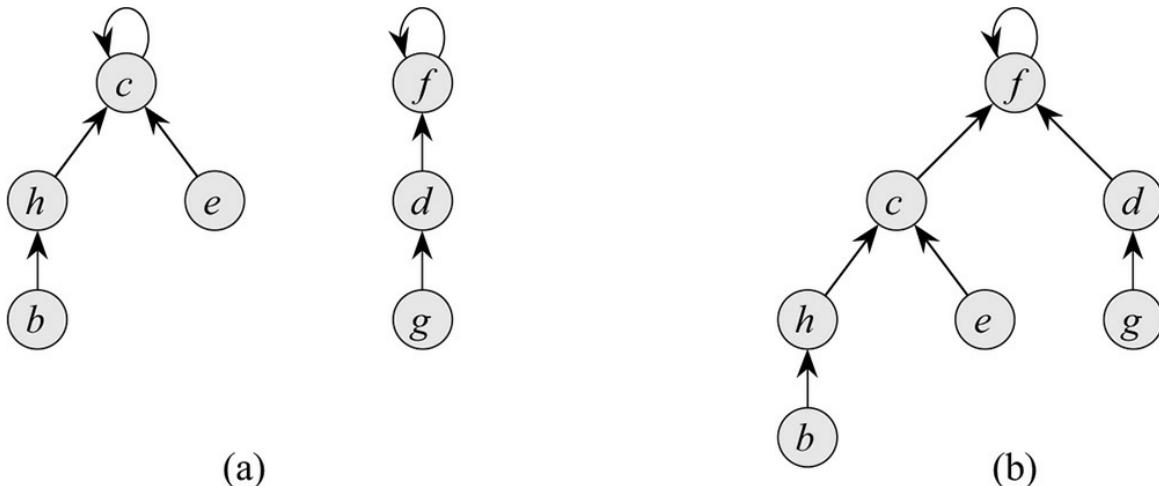
21.2-1 Escreva pseudocódigo para MAKE-SET, FIND-SET e UNION usando a representação de lista ligada e a heurística de união ponderada. Não esqueça de especificar os atributos que você considera para objetos conjunto e objetos lista.

21.2-2 Mostre a estrutura de dados resultante e as respostas retornadas pelas operações FIND-SET no programa a seguir. Use a representação de lista ligada com a heurística de união ponderada.

1 **for** $i = 1$ **to** 16
2 MAKE-SET(x_i)
3 **for** $i = 1$ **to** 15 **by** 2
4 UNION(x_i , x_{i+1})
5 **for** $i = 1$ **to** 13 **by** 4
6 UNION(xi , x_{i+2})
7 UNION(x_1, x_5)
8 UNION(x_{11}, x_{13})
9 UNION(x_1, x_{10})
10 FIND-SET(x_2)
11 FIND-SET(x_9)

Suponha que, se os conjuntos que contêm x_i e x_j tiverem o mesmo tamanho, a operação UNION(x_i, x_j) anexa a lista de x_j à lista de x_i.

21.2-3 Adapte a prova agregada do Teorema 21.1 para obter limites de tempo amortizados $O(1)$ para MAKE-SET e FIND-SET, e $O(\lg n)$ para UNION, utilizando a representação de lista ligada e a heurística de união ponderada.

21.2-4 Dê um limite assintótico restrito para o tempo de execução da sequência de operações na Figura 21.3, considerando a representação de lista ligada e a heurística de união ponderada.

21.2-5 O professor Gompers suspeita de que poderia ser possível manter apenas um ponteiro em cada objeto conjunto, em vez de dois (*head e tail*), e ao mesmo tempo manter em dois o número de ponteiros em cada elemento de lista. Mostre que a suspeita do professor é bem fundamentada descrevendo como representar cada conjunto por uma lista ligada de modo que cada operação tenha o mesmo tempo de execução que as operações descritas nesta seção. Descreva também como as operações funcionam. Seu esquema deve levar em consideração a heurística da união ponderada, com o mesmo efeito descrito nesta seção. (*Sugestão*: Use tail de uma lista ligada como seu representante de conjunto.)

21.2-6 Sugira uma mudança simples no procedimento UNION para a representação de lista ligada que elimine a necessidade de manter o ponteiro *tail* para o último objeto em cada lista. Independentemente de a heurística

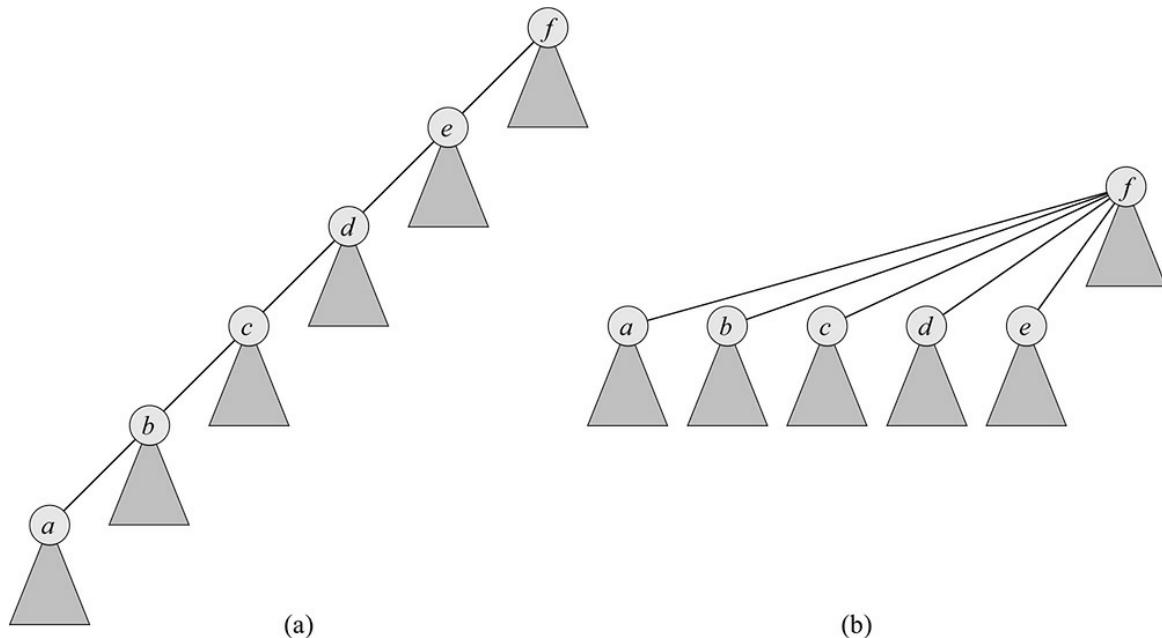
de união ponderada ser ou não utilizada, a alteração não deve mudar o tempo de execução assintótico do procedimento UNION. (*Sugestão*: Em vez de anexar uma lista à outra, entrelace uma à outra.)

21.3 FLORESTAS DE CONJUNTOS DISJUNTOS

Em uma implementação mais rápida de conjuntos disjuntos, representamos conjuntos por árvores enraizadas, sendo que cada nó contém um membro e cada árvore representa um conjunto. Em uma ***floresta de conjuntos disjuntos***, ilustrada na Figura 21.4(a), cada membro aponta apenas para seu pai. A raiz de cada árvore contém o representante e é seu próprio pai. Como veremos, embora os algoritmos diretos que utilizam essa representação não sejam mais rápidos que aqueles que usam a representação de lista ligada, com a introdução de duas heurísticas — "união pelo posto" e "compressão de caminho" — podemos obter uma estrutura de dados de conjuntos disjuntos assintoticamente ótima.

Figura 21.4 Uma floresta de conjuntos disjuntos. (**a**) Duas árvores que representam os dois conjuntos da Figura 21.2. A árvore à esquerda representa o conjunto $\{b, c, e, h\}$, com c como representante, e a árvore à direita representa o conjunto $\{d, f, g\}$, com f como representante. (**b**) Resultado de UNION(e, g).

Executamos as três operações de conjuntos disjuntos como descrevemos a seguir. Uma operação MAKE-SET simplesmente cria uma árvore com apenas um nó. Executamos uma operação FIND-SET seguindo ponteiros de pais até encontrarmos a raiz da árvore. Os nós visitados nesse caminho simples em direção à raiz constituem o ***caminho de localização***. Uma operação UNION, mostrada na Figura 21.4(b), faz a raiz de uma árvore apontar para a raiz da outra.

Heurísticas para melhorar o tempo de execução

Até agora, não melhoramos a implementação de listas ligadas. Uma sequência de n - 1 operações UNION pode criar uma árvore que é apenas uma cadeia linear de n nós. Contudo, usando duas heurísticas, podemos conseguir um tempo de execução quase linear em relação ao número total de operações m.

A primeira heurística, ***união pelo posto***, é semelhante à heurística de união ponderada que usamos com a representação de listas ligadas. A abordagem óbvia seria fazer a raiz da árvore que tem um número menor de nós apontar para a raiz da árvore que tem mais nós. Em vez de controlar explicitamente o tamanho da subárvore com raiz em cada nó, usaremos uma abordagem que facilita a análise. Para cada nó, mantemos um ***posto*** **(**rank**)** que é um limite superior para a altura do nó. Na união pelo posto, fazemos a raiz que ocupa o menor posto apontar para a raiz que ocupa o maior posto durante uma operação UNION.

A segunda heurística, ***compressão de caminho***, também é bastante simples e muito eficiente. Como mostra a Figura 21.5, nós a usamos durante operações FIND-SET para fazer cada nó no caminho de localização apontar diretamente para a raiz. A compressão de caminho não altera nenhum posto.

Pseudocódigo para florestas de conjuntos disjuntos

Para implementar uma floresta de conjuntos disjuntos com a heurística de união pelo posto, temos de controlar os postos. Com cada nó x, mantemos o valor $x.rank$, um inteiro, que é um limite superior para a altura de x (o número de arestas no caminho mais longo entre uma folha descendente e x). Quando MAKE-SET cria um conjunto unitário, o posto inicial do nó único na árvore correspondente é 0. As operações FIND-SET não alteram os postos. A operação UNION tem dois casos, dependendo de as raízes das árvores terem postos iguais ou não. Se as raízes têm postos desiguais, transformamos a raiz de posto mais alto no pai da raiz de posto mais baixo, mas os postos em si permanecem inalterados. Se, em vez disso, as raízes têm postos iguais, escolhemos arbitrariamente uma das raízes como o pai e incrementamos seu posto.

Figura 21.5 Compressão de caminho durante a operação FIND-SET. Setas e autolaços nas raízes foram omitidos. (**a**) Uma árvore que representa um conjunto antes da execução de FIND-SET(a). Triângulos representam subárvores cujas raízes são os nós mostrados. Cada nó tem um ponteiro para seu pai. (**b**) O mesmo conjunto após a execução de FIND-SET(a). Agora, cada nó no caminho de localização aponta diretamente para a raiz.

Vamos representar esse método em pseudocódigo. Designamos o pai do nó x por $x.p$. O procedimento LINK, uma sub-rotina chamada por UNION, adota ponteiros para duas raízes como entradas.

```
MAKE-SET(x)
1    x.p = x
2    x.rank = 0

UNION(x, y)
1    LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1    if x.rank > y.rank
2         y.p = x
3    else p[x] = y
4         if x.rank = y.rank
5              y.rank = y.rank+ 1
```

O procedimento FIND-SET com compressão de caminho é bastante simples.

```
FIND-SET(x)
1    if x ≠ x.p
2         x.p = FIND-SET(x.p)
3    return x.p
```

FIND-SET(x)

O procedimento FIND-SET é um ***método de duas passagens***: quando executa recursão, faz uma passagem para cima no caminho de localização para encontrar a raiz; à medida que a recursão se desenvolve, faz uma segunda passagem de volta para baixo no caminho de localização para atualizar cada nó de modo que cada um aponte diretamente para a raiz. Cada chamada de FIND-SET(x) retorna $p[x]$ na linha 3. Se x é a raiz, FIND-SET salta a linha 2 e, em vez de executá-la, retorna $x.p$, que é x. Esse é o caso em que a recursão chega ao nível mais baixo e recomeçaria novamente. Caso contrário, a linha 2 é executada, e a chamada recursiva com parâmetro $p[x]$ retorna um ponteiro para a raiz. A linha 2 atualiza o nó x para apontar diretamente para a raiz, e a linha 3 retorna esse ponteiro.

Efeito das heurísticas sobre o tempo de execução

Separadamente, união pelo posto ou compressão de caminho melhora o tempo de execução das operações em florestas de conjuntos disjuntos, e a melhora é ainda maior quando usamos as duas heurísticas em conjunto. Sozinha, a união pelo posto produz um tempo de execução de $O(m \lg n)$ (veja o Exercício 21.4-4), e esse limite é justo (veja o Exercício 21.3-3). Embora não o demonstremos aqui, para uma sequência de n operações MAKE-SET (e, consequentemente, no máximo n - 1 operações UNION) e f operações FIND-SET, a heurística de compressão de caminho, se sozinha, dá um tempo de execução do pior caso igual a $Q(n + f \cdot (1 + \log_{2 + f/n} n))$.

Quando usamos a união pelo posto e a compressão de caminho, o tempo de execução do pior caso é $O(m\ \alpha(n))$, onde $\alpha(n)$ é uma função de crescimento *muito* lento que definiremos na Seção 21.4. Em qualquer aplicação concebível de uma estrutura de dados de conjuntos disjuntos, $\alpha(n) \leq 4$; assim, podemos considerar o tempo de execução como linear em relação a m em todas as situações práticas. Na Seção 21.4, provaremos esse limite superior.

Exercícios

21.3-1 Faça novamente o Exercício 21.2-2 usando uma floresta de conjuntos disjuntos com união pelo posto e compressão de caminho.

21.3-2 Escreva uma versão não recursiva de FIND-SET com compressão de caminho.

21.3-3 Dê uma sequência de m operações MAKE-SET, UNION e FIND-SET, na qual n são operações MAKE-SET, que demore o tempo $(m \lg n)$ quando usamos somente união pelo posto.

21.3-4 Suponha que queiramos acrescentar a operação PRINT-SET(x), à qual é dado um nó x e que imprime todos os membros do conjunto de x em qualquer ordem. Mostre como podemos acrescentar apenas um único atributo a cada nó em uma floresta de conjuntos disjuntos de modo que PRINT-SET(x) demore tempo linear em relação ao número de membros do conjunto de x e que os tempos de execução assintóticos das outras operações permaneçam inalterados. Suponha que podemos imprimir cada membro do conjunto no tempo $O(1)$.

21.3-5 ★ Mostre que qualquer sequência de m operações MAKE-SET, FIND-SET e LINK, em que todas as operações LINK aparecem antes de qualquer das operações FIND-SET, demora apenas o tempo $O(m)$ se usarmos compressão de caminho e união pelo posto. O que acontece na mesma situação se usarmos somente a heurística de compressão de caminho?

21.4 ★ ANÁLISE DA UNIÃO PELO POSTO COM COMPRESSÃO DE CAMINHO

Como observamos na Seção 21.3, o tempo de execução da heurística combinada de união pelo posto e compressão de caminho é $O(m\ \alpha(n))$ para m operações de conjuntos disjuntos em n elementos. Nesta seção, examinaremos a função α para ver exatamente com que lentidão ela cresce. Então, provaremos esse tempo de execução empregando o método do potencial de análise amortizada.

Uma função de crescimento muito rápido e sua inversa de crescimento muito lento

Para inteiros $k \geq 0$ e $j \geq 1$, definimos a função $A_k(j)$ como

$$A_k(j) = \begin{cases} j+1 & \text{se } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{se } k \geq 1, \end{cases}$$

onde a expressão $A_{k-1}^{(j+1)}(j)$ usa a notação de iteração funcional dada na Seção 3.2. Especificamente, $A_{k-1}^{(0)}(j) = j$ e $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$ para $i \geq 1$. Faremos referência ao parâmetro k como o ***nível*** da função A.

A função $A_k(j)$ aumenta estritamente com j e k. Para ver exatamente com que rapidez essa função cresce, primeiro obtemos expressões de forma fechada para $A_1(j)$ e $A_2(j)$.

Lema 21.2

Para qualquer inteiro $j \geq 1$, temos $A_1(j) = 2j + 1$.

Prova Primeiro usamos indução em i para mostrar que $A_1^{(i)}(j) = 2^i(j + 1) - 1$. Para o caso básico, temos $A_1^{(0)}(j) = j = 2^0(j + 1) - 1$. Para o passo de indução, suponha que $A_1^{(i-1)}(j) = 2^{i-1}(j + 1) - 1$. Então, $A_1^{(i)}(j) = A_1(A_1^{i-1}(j)) = A_1(2^{i-1}(j + 1) - 1) = 2{\cdot}(2^{i-1}(j + 1) - 1) + 1 = 2^i(j + 1) - 2 + 1 = 2^i(j + 1) - 1$. Finalmente, observamos que $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j + 1) - 1$. ■

Lema 21.3

Para qualquer inteiro j 1, temos $A_2(j) = 2^{j+1}(j + 1) - 1$.

Prova Primeiro usamos indução em i para mostrar que $A_1^{(i)}(j) = 2^i(j + 1) - 1$. Para o caso básico, temos $A_1^{(0)}(j) = j = 2^0(j + 1) - 1$. Para o passo de indução, suponha que $A_1^{(i-1)}(j) = 2^{i-1}(j + 1) - 1$. Então, $A_1^{(i)}(j) = A_1(A_1^{i-1}(j)) = A_1(2^{i-1}(j + 1) - 1) = 2{\cdot}(2^{i-1}(j + 1) - 1) + 1 = 2^i(j + 1) - 2 + 1 = 2^i(j + 1) - 1$. Finalmente, observamos que $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j + 1) - 1$. ■

Agora podemos ver com que rapidez $A_k(j)$ cresce, simplesmente examinando $A_k(1)$ para os níveis $k = 0, 1, 2, 3, 4$. Pela definição de $A_0(k)$ e lemas citados, temos $A_0(1) = 1 + 1 = 2$, $A_1(1) = 2 \cdot 1 + 1 = 3$ e $A_2(1) = 2^{1+1} \cdot (1 + 1) - 1 = 7$. Também temos

$$\begin{aligned}
A_3(1) &= A_2^{(2)}(1) \\
&= A_2(A_2(1)) \\
&= A_2(7) \\
&= 2^8 \cdot 8 - 1 \\
&= 2^{11} - 1 \\
&= 2047
\end{aligned}$$

e

$$\begin{aligned}
A_4(1) &= A_3^{(2)}(1) \\
&= A_3(A_3(1)) \\
&= A_2^{(2048)}(2047) \\
&\gg A_2(2047) \\
&= 2^{2048} \cdot 2048 - 1 \\
&> 2^{2048} \\
&= (2^4)^{512} \\
&= 16^{512} \\
&\gg 10^{80},
\end{aligned}$$

que é o número estimado de átomos no universo observável. (O símbolo “” denota a relação “muito maior que”.)

Definimos a inversa da função $A_k(n)$, para inteiros $n \geq 0$, por

$$\alpha(n) = \min \{k : A_k(1) \geq n\} .$$

Em linguagem corrente, $\alpha(n)$ é o mais baixo nível k para o qual $A_k(1)$ é no mínimo n. Pelos valores de $A_k(1)$, vemos que

$$\alpha(n) = \begin{cases} 0 & \text{para } 0 \leq n \leq 2, \\ 1 & \text{para } n = 3, \\ 2 & \text{para } 4 \leq n \leq 7, \\ 3 & \text{para } 8 \leq n \leq 2047, \\ 4 & \text{para } 2048 \leq n \leq A_4(1). \end{cases}$$

É apenas para valores de n tão grandes que nem mesmo o termo “astronômico” dá ideia de seu tamanho (maior do que $A_4(1)$, um número enorme) que $\alpha(n) > 4$. Portanto, $\alpha(n) \leq 4$ para todas as finalidades práticas.

Propriedades de posto

No restante desta seção, provaremos um limite $O(m\ \alpha(n))$ para o tempo de execução das operações de conjuntos disjuntos com união pelo posto e compressão de caminho. Para provar esse limite, primeiro demonstraremos algumas propriedades simples de posto.

Lema 21.4

Para todos os nós x, temos $x.rank \leq x.p.rank$, com desigualdade estrita se $x \neq x.p$. O valor de $x.rank$ é inicialmente 0 e aumenta com o tempo até $x \neq p[x]$; daí em diante, $x.rank$ não muda. O valor de $x.p.rank$ cresce monotonicamente com o tempo.

Prova A prova é uma indução direta em relação ao número de operações, com a utilização das implementações de MAKE-SET, UNION e FIND-SET que aparecem na Seção 21.3. Vamos deixá-la para o Exercício 21.4-1.

Corolário 21.5

À medida que seguimos o caminho simples de qualquer nó em direção a uma raiz, os postos dos nós aumentam estritamente.

Lema 21.6

Todo nó tem posto, no máximo, igual a n - 1.

Prova O posto de cada nó começa em 0 e só aumenta com operações LINK. Como há no máximo n - 1 operações UNION, há também no máximo n - 1 operações LINK. Como cada operação LINK deixa todas os postos como estão ou aumenta de 1 o posto de algum nó, todos os postos são no máximo n - 1.

O Lema 21.6 dá um limite fraco para postos. De fato, todo nó tem posto no máximo lg n (veja o Exercício 21.4-2). Todavia, o limite mais frouxo do Lema 21.6 será suficiente para a nossa finalidade.

Provando o limite de tempo

Usaremos o método do potencial da análise amortizada (veja a Seção 17.3) para provar o limite de tempo $O(m\ \alpha(n))$. Ao executarmos a análise amortizada, é conveniente considerar que invocamos a operação LINK em vez da operação UNION. Isto é, visto que os parâmetros do procedimento LINK são ponteiros para duas raízes, agimos como se executássemos as operações FIND-SET adequadas separadamente. O lema a seguir mostra que, ainda que contemos as operações FIND-SET extras induzidas por chamadas UNION, o tempo de execução assintótico permanece inalterado.

Lema 21.7

Suponha que convertemos uma sequência S' de m' operações MAKE-SET, UNION e FIND-SET em uma sequência S de m operações MAKE-SET, LINK e FIND-SET transformando cada UNION em duas operações FIND-SET seguidas por uma operação LINK. Então, se a sequência S for executada no tempo $O(m\ \alpha(n))$, a sequência S' será executada no tempo $O(m'\ \alpha(n))$.

Prova Visto que cada operação UNION na sequência S' é convertida em três operações em S, temos $m' \leq m \leq 3m'$. Como $m = O(m')$, um limite de tempo $O(m\ \alpha(n))$ para a sequência convertida S implica um limite de tempo $O(m'\ \alpha(n))$ para a sequência original S'.

No restante desta seção, supomos que a sequência inicial de m' operações MAKE-SET, UNION e FIND-SET foi convertida em uma sequência de m operações MAKE-SET, LINK e FIND-SET. Agora, provaremos um limite de tempo $O(m\ \alpha(n))$ para a sequência convertida e apelaremos para o Lema 21.6 para provar o tempo de execução $O(m'\ \alpha(n))$ da sequência original de m' operações.

Função potencial

A função potencial que usamos atribui um potencial $f_q(x)$ a cada nó x na floresta de conjuntos disjuntos após q operações. Somamos os potenciais de nós para obter o potencial da floresta inteira: $F_q = Sx, f_q(x)$, onde F_q denota o potencial da floresta após q operações. A floresta está vazia antes da primeira operação, e definimos arbitrariamente $F_0 = 0$. Jamais algum potencial F_q será negativo.

O valor de $f_q(x)$ depende de x ser uma raiz da árvore após a q-ésima operação. Se for ou se $x.rank = 0$, então $f_q(x) = \alpha(n) \cdot x.rank$.

Agora, suponha que após a q-ésima operação x não seja uma raiz e $x.rank \geq 1$. Precisamos definir duas funções auxiliares em x antes de podermos definir $F_q(x)$. Primeiro, definimos

$$\text{nível}(x) = \max \{k : x.p.rank \geq A_k(x.rank)\}\ .$$

Isto é, nível(x) é o maior nível k para o qual A_k, aplicada à rank x, não é maior que o posto do pai de x.

Afirmamos que

$$0 \leq \text{nível}(x) < \alpha(n)\ , \qquad (21.1)$$

o que vemos a seguir. Temos

$$\begin{aligned} x.p.rank &\geq x.rank + 1 && \text{(pelo Lema 21.4)} \\ &= A_0(x.rank) && \text{(pela definição de } A_0(j)) , \end{aligned}$$

o que implica que nível(x) ≥ 0, e temos

$$\begin{aligned} A_{\alpha(n)}(x.rank) &\geq A_{\alpha(n)}(1) && \text{(porque } A_k(j) \text{ é estritamente crescente)} \\ &\geq n && \text{(pela definição de } \alpha(n)) \\ &> x.p.rank && \text{(pelo Lema 21.6)} , \end{aligned}$$

o que implica que nível(x) < $\alpha(n)$. Observe que, em razão de $x.p.rank$ aumentar monotonicamente com o tempo, o mesmo ocorre com nível(x).

A segunda função auxiliar se aplica quando $x.rank \geq 1$:

$$\text{iter}(x) = \max \{i : x.p.rank \geq A^{(i)}_{\text{nível}(x)}(x.rank)\}.$$

Isto é, iter(x) é o maior número de vezes que podemos aplicar a $A_{\text{nível}(x)}$, iterativamente, aplicada inicialmente ao posto de x, antes de obtermos um valor maior que o posto do pai de x.

Afirmamos que, quando $x.rank \geq 1$, temos

$$1 \leq \text{iter}(x) \leq x.rank , \tag{21.2}$$

o que mostramos a seguir. Temos

$$\begin{aligned} x.p.rank &\geq A_{\text{nível}(x)}(x.rank) && \text{(pela definição de nível}(x)) \\ &= A^{(1)}_{\text{nível}(x)}(x.rank) && \text{(pela definição de iteração funcional)} , \end{aligned}$$

o que implica que iter(x) ≥ 1, e temos

$$\begin{aligned} A^{(x.rank+1)}_{\text{nível}(x)}(x.rank) &= A_{\text{nível}(x)+1}(x.rank) && \text{(pela definição de } A_k(j)) \\ &> x.p.rank && \text{(pela definição de nível}(x) , \end{aligned}$$

o que implica que iter(x) ≤ $x.rank$. Observe que, como $x.p.rank$ cresce monotonicamente com o tempo, para iter(x) diminuir, nível(x) tem de aumentar. Contanto que nível(x) permaneça inalterado, iter(x) deve crescer ou permanecer inalterado.

Estabelecidas essas funções auxiliares, estamos prontos para definir o potencial do nó x após q operações:

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot x.rank & \text{if } x \text{ is a root or } x.rank = 0, \\ (\alpha(n) - \text{nível(x)} \cdot x.rank - iter(x) & \text{if } x \text{ is not a root and } x.rank \geq 1. \end{cases}$$

Em seguida, investigamos algumas propriedades úteis desses potenciais de nós.

Lema 21.8

Para todo nó x, e para todas as contagens de operações q, temos $0 \leq q(x) \leq \alpha(n) \cdot x.rank$.

$$0 \leq \phi_q(x) \leq \alpha(n) \cdot x.rank .$$

Prova Se x é uma raiz ou $x.rank = 0$, então $q(x) = \alpha(n) \cdot x.rank$ por definição. Agora, suponha que x não seja uma raiz e que $x.rank \geq 1$. Obtemos um limite inferior para $q(x)$ maximizando nível(x) e iter(x). Pelo limite (21.1), nível(x) ≤ $\alpha(n)$ - 1 e, pelo limite (21.2), iter(x) ≥ $x.rank$. Assim,

$$
\begin{aligned}
\phi_q(x) &= (\alpha(n) - \text{nível}(x)) \cdot x.rank - \text{iter}(x) \\
&\geq (\alpha(n) - (\alpha(n) - 1)) \cdot x.rank - x.rank \\
&= x.rank - x.rank \\
&= 0.
\end{aligned}
$$

De modo semelhante, obtemos um limite superior para $q(x)$ minimizando nível(x) e iter(x). Pelo limite (21.1), nível(x) $\geq$ 0 e, pelo limite (21.2), iter(x) $\geq$ 1. Portanto,

$$
\begin{aligned}
\phi_q(x) &\leq (\alpha(n) - 0) \cdot x.rank - 1 \\
&= \alpha(n) \cdot x.rank - 1 \\
&< \alpha(n) \cdot x.rank.
\end{aligned}
$$

Corolário 21.9

Se o nó x não é uma raiz e $x.rank > 0$, então $f_q(x) < \alpha(n) \cdot x.rank$.

Mudanças de potencial e custos amortizados de operações

Agora, estamos prontos para examinar como as operações de conjuntos disjuntos afetam os potenciais de nós. Se entendermos a mudança de potencial provocada por cada operação, poderemos determinar o custo amortizado de cada operação.

Lema 21.10

Seja x um nó que não é uma raiz, e suponha que a q-ésima operação é LINK ou FIND-SET. Então, após a q-ésima operação, $f_q(x) \leq f_{q-1}(x)$. Além disso, se $x.rank \geq 1$ e nível(x) ou iter(x) mudar devido à q-ésima operação, então $f_q(x) \leq f_{q-1}(x) - 1$. Isto é, o potencial de x não pode aumentar e, se tiver posto positivo e nível(x) ou iter(x) mudar, o potencial de x cairá no mínimo uma unidade.

Prova Como x não é uma raiz, a q-ésima operação não muda $x.rank$ e, como n não muda após as n operações MAKE-SET iniciais, $\alpha(n)$ também permanece inalterada. Consequentemente, esses componentes da fórmula para o potencial de x permanecem os mesmos após a q-ésima operação. Se $x.rank = 0$, então $f_q(x) = f_{q-1}(x) = 0$. Agora, suponha que $x.rank \geq 1$. Lembre-se de que nível(x) aumenta monotonicamente com o tempo. Se a q-ésima operação deixar nível(x) inalterado, iter(x) aumenta ou permanece inalterado. Se nível(x) e iter(x) permanecem inalterados, $f_q(x) = f_{q-1}(x)$. Se nível(x) permanece inalterado e iter(x) aumenta, então ele aumenta, no mínimo, de 1, e assim $f_q(x) \leq f_{q-1}(x) - 1$. Finalmente, se a q-ésima operação aumentar nível(x), ele aumentará no mínimo de 1, de modo que o valor do termo $(\alpha(n)$ - nível(x)) $\cdot$ $x.rank$ cairá no mínimo $x.rank$. Como nível(x) aumentou, o valor de iter(x) poderá cair, mas, de acordo com o limite (21.2), a queda é no máximo $x.rank - 1$. Assim, o aumento de potencial devido à mudança em iter(x) é menor que a queda de potencial devido à mudança em nível(x), e concluímos que $f_q(x) \leq f_{q-1}(x) - 1$.

Nossos três lemas finais mostram que o custo amortizado de cada operação MAKE-SET, LINK e FIND-SET é $O(\alpha(n))$. Lembre-se de que, pela equação (17.2), o custo amortizado de cada operação é seu custo real mais o aumento em potencial devido à operação.

Lema 21.11

O custo amortizado de cada operação MAKE-SET é $O(1)$.

Prova Suponha que a q-ésima operação seja MAKE-SET(x). Essa operação cria o nó x com posto 0, de modo que $f_q(x) = 0$. Nenhum outro posto ou potencial se altera, e então $F_q = F_{q-1}$. Observar que o custo real da operação MAKE-SET é $O(1)$ conclui a prova.

Lema 21.12

O custo amortizado de cada operação LINK é $O(\alpha(n))$.

Prova Suponha que a q-ésima operação seja LINK(x, y). O custo real da operação LINK é $O(1)$. Sem prejuízo da generalidade, suponha que a operação LINK torne y o pai de x. Para determinar a mudança de potencial devido a LINK, observamos que os únicos nós cujos potenciais podem mudar são x, y e os filhos de y imediatamente antes da operação. Mostraremos que o único nó cujo potencial pode aumentar devido a LINK é y e que seu aumento é no máximo $\alpha(n)$:

- Pelo Lema 21.10, qualquer nó que seja filho de y imediatamente antes de LINK não pode ter seu aumento de potencial devido a LINK.
- Pela definição de $f_q(x)$, vemos que, como x era uma raiz antes da q-ésima operação, $f_{q-1}(x) = \alpha(n) \cdot x.rank$. Se $x.rank = 0$, então $f_q(x) = f_{q-1}(x) = 0$. Caso contrário,

$$\begin{aligned} \phi_q(x) &< \alpha(n) \cdot x.rank \quad \text{(pelo Corolário 21.9)} \\ &= \phi_{q-1}(x) \, , \end{aligned}$$

  e, assim, o potencial de x diminui.
- Como y é uma raiz antes de LINK, $f_{q-1}(y) = a(n) \cdot y.rank$. A operação LINK deixa y como raiz e deixa o posto de y como está ou aumenta o posto de y de 1. Assim, $f_q(y) = f_{q-1}(y)$ ou $f_q(y) = f_{q-1}(y) + \alpha(n)$.

Portanto, o aumento de potencial devido à operação LINK é no máximo $\alpha(n)$. O custo amortizado da operação LINK é $O(1) + \alpha(n) = O(\alpha(n))$.

Lema 21.13

O custo amortizado de cada operação FIND-SET é $O(\alpha(n))$.

Prova Suponha que a q-ésima operação seja FIND-SET e que o caminho de localização contenha s nós. O custo real da operação FIND-SET é $O(s)$. Mostraremos que nenhum potencial de nó aumenta devido a FIND-SET e que, no mínimo, $\max(0, s - (\alpha(n) + 2))$ nós no caminho de localização têm seu potencial diminuído de no mínimo 1.
Para ver que nenhum potencial de nó aumenta, primeiro apelamos para o Lema 21.9 para todos os nós exceto a raiz. Se x é a raiz, seu potencial é $\alpha(n) \cdot x.rank$, que não muda.
Agora, mostramos que $\max(0, s - (\alpha(n) + 2))$ nós têm seu potencial diminuído de no mínimo 1. Seja x um nó no caminho de localização, tal que $x.rank > 0$ e x é seguido em algum lugar no caminho de localização por outro nó y que não é uma raiz, onde nível(y) = nível(x) imediatamente antes da operação FIND-SET. (O nó y não precisa seguir x *imediatamente* no caminho de localização.) Todos os nós exceto, no máximo, $\alpha(n) + 2$ nós no caminho de localização satisfazem essas restrições para x. Os que não as satisfazem são o primeiro nó no caminho de localização (se ele tiver posto 0), o último nó no caminho (isto é, a raiz) e o último nó w no caminho para o qual nível(w) = k, para cada $k = 0, 1, 2, ..., \alpha(n) - 1$.
Vamos fixar tal nó x, e mostraremos que o potencial de x diminui de, no mínimo, 1. Seja k = nível(x) = nível(y). Imediatamente antes da compressão de caminho causada por FIND-SET, temos

$$\begin{aligned}
x.p.rank &\geq A_k^{(\text{iter}(x))}(x.rank) && \text{(pela definição de iter}(x)) , \\
y.p.rank &\geq A_k(y.rank) && \text{(pela definição de nível}(y)) , \\
y.rank &\geq x.p.rank && \text{(pelo Corolário 21.5 e porque } y \text{ segue } x \text{ no caminho de localização).}
\end{aligned}$$

Reunindo essas desigualdades e sendo i o valor de iter(x) antes da compressão de caminho, temos

$$\begin{aligned}
y.p.rank &\geq A_k(y.rank) \\
&\geq A_k(x.p.rank) && \text{(porque } A_k(j) \text{ é estritamente crescente)} \\
&\geq A_k(A_k^{(\text{inter}(x))}(x.rank)) \\
&= A_k^{(i+1)}(x.rank)) .
\end{aligned}$$

Como a compressão de caminho fará x e y terem o mesmo pai, sabemos que, após a compressão de caminho, $x.p.rank = y.p.rank$ e que a compressão de caminho não diminui $y.p.rank$. Visto que $x.rank$ não muda, após a compressão de caminho temos que $x.p.rank \geq A_k^{(i+1)}(x.rank))$. Assim, a compressão de caminho fará iter(x) aumentar (até no mínimo $i + 1$) ou nível(x) aumentar (o que ocorre se iter(x) aumentar até no mínimo $x.rank + 1$). Em qualquer caso, pelo Lema 21.10, temos $f_q(x) \leq f_{q-1}(x) - 1$. Consequentemente, o potencial de x diminui de no mínimo 1.
O custo amortizado da operação Find-Set é o custo real mais a mudança de potencial. O custo real é $O(s)$, e mostramos que o potencial total diminui de, no mínimo, $\max(0, s - (\alpha(n) + 2))$. Portanto, o custo amortizado é, no máximo, $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, já que podemos aumentar a escala das unidades de potencial para dominar a constante oculta em $O(s)$.

Reunindo os lemas precedentes, temos o teorema a seguir.

Teorema 21.14

Uma sequência de m operações Make-Set, Union e Find-Set, das quais n são operações Make-Set, pode ser executada em uma floresta de conjuntos disjuntos com união pelo posto e compressão de caminho no tempo do pior caso $O(m\,\alpha(n))$.

Prova Imediata, pelos Lemas 21.7, 21.11, 21.12 e 21.13.

Exercícios

21.4-1 Prove o Lema 21.4.

21.4-2 Prove que todo nó tem posto no máximo igual a $\lg n$.

21.4-3 De acordo com o Exercício 21.4-2, quantos bits são necessários para armazenar $x.rank$ >para cada nó x?

21.4-4 Usando o Exercício 21.4-2 dê uma prova simples de que operações em uma floresta de conjuntos disjuntos com união pelo posto, mas sem compressão de caminho, são executadas no tempo $O(m \lg n)$.

21.4-5 O professor Dante argumenta que, como os postos de nós aumentam estritamente ao longo de um caminho simples até a raiz, os níveis de nós devem aumentar monotonicamente ao longo do caminho. Em outras palavras, se $x.rank > 0$ e $x.p$ não é uma raiz, nível(x) $\leq$ nível($x.p$). O professor está correto?

21.4-6 ★ Considere a função $\alpha'(n) = \min\{k : A_k(1) \geq \lg(n + 1)\}$. Mostre que $\alpha'(n) \leq 3$ para todos os valores práticos de n e, usando o Exercício 21.4-2, mostre como modificar o argumento da função potencial para provar que podemos executar uma sequência de m operações Make-Set, Union e Find-Set, das quais n são

operações MAKE-SET, em uma floresta de conjuntos disjuntos com união pelo posto e compressão de caminho no tempo do pior caso $O(m\ \alpha'(n))$.

Problemas

21-1 Mínimo off-line

O problema do ***mínimo off-line*** nos pede para manter um conjunto dinâmico T de elementos do domínio $\{1, 2, ..., n\}$ sob as operações INSERT e EXTRACT-MIN. Temos uma sequência S de n chamadas INSERT e m chamadas EXTRACT-MIN, onde cada chave em $\{1, 2, ..., n\}$ é inserida exatamente uma vez. Desejamos determinar qual chave é retornada por cada chamada EXTRACT-MIN. Especificamente, desejamos preencher um arranjo $extraído[1 .. m]$, onde para $i = 1, 2, ..., m$, $extraído[i]$ é a chave retornada pela i-ésima chamada EXTRACT-MIN. O problema é "off-line" no sentido de que temos a possibilidade de processar a sequência S inteira antes de determinar quaisquer das chaves retornadas.

a. Na seguinte instância do problema do mínimo off-line, cada operação INSERT(i) é representada pelo valor de i, e cada EXTRACT-MIN é representada pela letra E:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.

Preencha os valores corretos no arranjo *extraído*.

Para desenvolver um algoritmo para esse problema, desmembramos a sequência S em subsequências homogêneas. Isto é, representamos S por

I_1, E, I_2, E, I_3, ..., I_m, E, I_{m+1} ,

onde cada E representa uma única chamada EXTRACT-MIN, e cada I_j representa uma sequência (possivelmente vazia) de chamadas INSERT. Para cada subsequência I_j, colocamos inicialmente as chaves inseridas por essas operações em um conjunto K_j, que é vazio se I_j é vazio. Em seguida, fazemos:

```
OFF-LINE-MINIMUM(m, n)
1 for i = 1 to n
2        determinar j tal que i ∈ K_j
3        if j ≠ m + 1
4                extraído[j] = i
5                seja l o menor valor maior que j
                         para o qual o conjunto K_l existe
6                K_l = K_j ∪ K_l, destruindo K_j
7 return extraído
```

b. Demonstre que o arranjo *extraído* retornado por OFF-LINE-MINIMUM é correto.

c. Descreva como implementar OFF-LINE-MINIMUM eficientemente com uma estrutura de dados de conjuntos disjuntos. Dê um limite justo para o tempo de execução do pior caso de sua implementação.

21-2 Determinação da profundidade

No ***problema de determinação da profundidade***, mantemos uma floresta $F = \{T_i\}$ de árvores enraizadas sob três operações:

MAKE-TREE(v) cria uma árvore cujo único nó é v.

FIND-DEPTH(v) retorna a profundidade do nó v dentro de sua árvore.

GRAFT(r, v) faz o nó r, que supomos ser a raiz de uma árvore, se tornar o filho do nó v, o qual supomos estar em uma árvore diferente de r, mas que pode ou não ser ele próprio uma raiz.

a. Suponha que utilizemos uma representação de árvore semelhante a uma floresta de conjuntos disjuntos: $v.p$ é o pai do nó v, exceto que $v.p = v$ se v é uma raiz. Suponha ainda mais que implementemos GRAFT(r, v), atribuindo $r.p = v$ e FIND-DEPTH(v) seguindo o caminho de localização até a raiz, devolvendo uma contagem de todos os nós encontrados, exceto v. Mostre que o tempo de execução no pior caso de uma sequência de m operações MAKE-TREE, FIND-DEPTH e GRAFT é Q(m_2).

Usando a heurística de união pelo posto e compressão de caminho, podemos reduzir o tempo de execução do pior caso. Usamos a floresta de conjuntos disjuntos S = $\{S_i\}$, onde cada conjunto S_i (que é ele próprio uma árvore) corresponde a uma árvore T_i na floresta F. Contudo, a estrutura de árvore dentro de um conjunto S_i não corresponde necessariamente à de T_i. De fato, a implementação de S_i não registra as relações pai-filho exatas mas, apesar disso, nos permite determinar a profundidade de qualquer nó em T_i.

A ideia fundamental é manter em cada nó v uma "pseudodistância" $v.d$, definida de tal modo que a soma das pseudodistâncias ao longo do caminho simples de v até a raiz de seu conjunto S_i seja igual à profundidade de v em T_i. Isto é, se o caminho simples de v até sua raiz em S_i é $v_0, v_1, \ldots v_k$, onde $v_0 = v$ e v_k é a raiz de S_i , então a profundidade de v em T_i é $\sum_{j=0}^{k} v_j \cdot d.$

b. Dê uma implementação de MAKE-TREE.

c. Mostre como modificar FIND-SET para implementar FIND-DEPTH. Sua implementação deve executar compressão de caminho e seu tempo de execução deve ser linear em relação ao comprimento do caminho de localização. Certifique-se de que sua implementação atualiza pseudodistâncias corretamente.

d. Mostre como implementar GRAFT(r, v), que combina os conjuntos que contêm r e v, modificando os procedimentos UNION e LINK. Certifique-se de que sua implementação atualize pseudodistâncias corretamente. Observe que a raiz de um conjunto S_i não é necessariamente a raiz da árvore correspondente T_i.

e. Dê um limite justo para o tempo de execução do pior caso de uma sequência de m operações MAKE-TREE, FIND-DEPTH e GRAFT, das quais n são operações MAKE-TREE.

21-3 Algoritmo off-line de Tarjan para o menor ancestral comum

O ***menor ancestral comum*** de dois nós u e v em uma árvore enraizada T é o nó w que é um ancestral de u e v e que tem a maior profundidade em T. No ***problema dos menores ancestrais comuns off-line***, temos uma árvore enraizada T e um conjunto arbitrário $P = \{\{u, v\}\}$ de pares não ordenados de nós em T e desejamos determinar o menor ancestral comum de cada par em P.

Para resolver o problema dos menores ancestrais comuns off-line, o procedimento a seguir executa um percurso da árvore T com a chamada inicial LCA($T.raiz$). Supomos que cada nó tem a cor WHITE antes do percurso.

```
LCA(u)
1     MAKE-SET(u)
2     FIND-SET(u).ancestral = u
3     for cada filho v de u em T
4         LCA(v)
5         UNION(u, v)
6         FIND-SET(u).ancestral = u
7     u.cor = BLACK
8     for cada nó v tal que {u, v} ∈ P
9         if v.cor == BLACK
10            imprimir "O menor ancestral comum de"
                  u "e" v "é" FIND-SET(u).ancestral
```

a. Demonstre que a linha 10 é executada exatamente uma vez para cada par $\{u, v\} \in P$.

b. Demonstre que, no momento da chamada LCA(u), o número de conjuntos na estrutura de dados de conjuntos disjuntos é igual à profundidade de u em T.

c. Prove que LCA imprime corretamente o menor ancestral comum de u e v para cada par $\{u, v\} \in P$.

d. Analise o tempo de execução de LCA considerando que usamos a implementação da estrutura de dados de conjuntos disjuntos da Seção 21.3.

NOTAS DO CAPÍTULO

Muitos dos resultados importantes para estruturas de dados de conjuntos disjuntos se devem em parte a R. E. Tarjan. Usando análise agregada, Tarjan [328, 330] deu o primeiro limite superior restrito em termos da inversa $\hat{\alpha}(m, n)$ de crescimento muito lento da função de Ackermann. (A função $A_k(j)$ dada na Seção 21.4 é semelhante à função de Ackermann, e a função $\alpha(n)$ é semelhante à inversa. Ambas $\alpha(n)$ e $\hat{\alpha}(m, n)$ são no máximo 4 para todos os valores concebíveis de m e n.) Um limite superior $O(m \lg^* n)$ foi provado antes por Hopcroft e Ullman [5, 179]. O tratamento na Seção 21.4 foi adaptado de uma análise posterior de Tarjan [332] que, por sua vez, é baseada em uma análise de Kozen [220]. Harfst e Reingold [161] dão uma versão baseada em potencial do limite anterior de Tarjan.

Tarjan e van Leeuwen [333] discutem variantes da heurística de compressão de caminho, inclusive "métodos de uma passagem" que, às vezes, oferecem melhores fatores constantes em seu desempenho que os métodos de duas passagens. Assim como as primeiras análises da heurística básica de compressão de caminho realizadas apenas por Tarjan, as realizadas por Tarjan e Leeuwen são agregadas. Mais tarde, Harfst e Reingold [161] mostraram como fazer uma pequena mudança na função potencial para adaptar sua análise de compressão de caminho a essas variantes de uma passagem. Gabow e Tarjan [121] mostram que, em certas aplicações, é possível fazer com que as operações em conjuntos disjuntos sejam executadas no tempo $O(m)$.

Tarjan [329] mostrou que um limite inferior de tempo $(m\,\hat{\alpha}(m, n))$ é exigido para operações em qualquer estrutura de dados de conjuntos disjuntos que satisfaça certas condições técnicas. Mais tarde, esse limite inferior foi generalizado por Fredman e Saks [113], que mostraram que, no pior caso, $(m\hat{\alpha}(m, n))$ palavras de $(\lg n)$ bits devem ser acessadas.

---

[1] Quando as arestas do grafo são estáticas — não mudam com o tempo —, podemos calcular as componentes conexas mais rapidamente usando busca em profundidade (Exercício 22.3-12). Contudo, às vezes, as arestas são acrescentadas dinamicamente e precisamos manter as componentes conexas à medida que cada aresta é acrescentada. Nesse caso, a implementação dada aqui pode ser mais eficiente do que executar uma nova busca primeiro em profundidade para cada nova aresta.

Parte

ALGORITMOS DE GRAFOS

INTRODUÇÃO

Problemas sobre grafos estão sempre presentes em ciência da computação, e algoritmos para trabalhar com eles são fundamentais para a área. Centenas de problemas computacionais interessantes são expressos em termos de grafos. Nesta parte, examinaremos alguns dos mais significativos.

O Capítulo 22 mostra como podemos representar um grafo em um computador e depois discute os algoritmos baseados na pesquisa de um grafo utilizando busca em largura ou busca em profundidade. O capítulo apresenta duas aplicações de busca em profundidade: ordenação topológica de um grafo acíclico dirigido e decomposição de um grafo dirigido em suas componentes fortemente conexas.

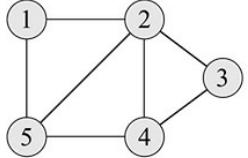
O Capítulo 23 descreve como calcular uma árvore geradora de peso mínimo de um grafo: o modo de menor peso para conectar todos os vértices quando cada aresta tem um peso associado. Os algoritmos para calcular árvores geradoras mínimas são bons exemplos de algoritmos gulosos (veja o Capítulo 16).

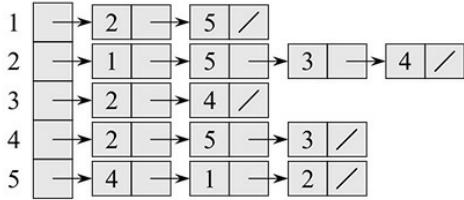
Os Capítulos 24 e 25 consideram como calcular caminhos mínimos entre vértices quando cada aresta tem um comprimento ou "peso" associado. O Capítulo 24 mostra como encontrar caminhos mínimos de determinado vértice de fonte até todos os outros vértices, e o Capítulo 25 examina métodos para calcular caminhos mínimos entre cada par de vértices.

Finalmente, o Capítulo 26 mostra como calcular um fluxo máximo de material em uma rede de fluxo, que é um grafo dirigido que tem um vértice de fonte de material especificado, um vértice sorvedouro especificado e capacidades especificadas para a quantidade de material que pode percorrer cada aresta dirigida. Esse problema geral surge sob muitas formas, e um bom algoritmo para calcular fluxos máximos pode ajudar a resolver eficientemente uma variedade de problemas relacionados.

Quando caracterizamos o tempo de execução de um algoritmo de grafo em um determinado grafo $G = (V, E)$, normalmente medimos o tamanho da entrada em termos do número de vértices $|V|$ e do número de arestas $|E|$ do grafo. Isto é, descrevemos o tamanho da entrada com dois parâmetros, não apenas um. Adotamos uma convenção de notação comum para esses parâmetros. Dentro da notação assintótica (como a notação O ou a notação Q), e *somente* dentro de tal notação, o símbolo V denota $|V|$ e o símbolo E denota $|E|$. Por exemplo, poderíamos dizer que "o algoritmo é executado em tempo $O(VE)$", significando que o algoritmo é executado no tempo $O(|V||E|)$. Essa convenção torna as fórmulas de tempo de execução mais fáceis de ler, sem risco de ambiguidade.

Outra convenção que adotamos aparece no pseudocódigo. Denotamos o conjunto de vértices de um grafo G por $G.V$ e seu conjunto de arestas por $G.E$. Isto é, o pseudocódigo vê os conjuntos de vértices e arestas como atributos de um grafo.

22 ALGORITMOS ELEMENTARES EM GRAFOS

Este capítulo apresenta métodos para representar um grafo e para executar busca em um grafo. Executar busca em um grafo significa seguir sistematicamente as arestas do grafo de modo a visitar os vértices do grafo. Um algoritmo de busca pode revelar muita coisa sobre a estrutura de um grafo. Muitos algoritmos começam executando uma busca em seu grafo de entrada para obter essas informações estruturais. Vários outros algoritmos trabalham em cima de uma busca básica em grafos. As técnicas para executar busca em um grafo estão no núcleo da área de algoritmos em grafos.

A Seção 22.1 discute as duas representações computacionais mais comuns de grafos: como listas de adjacências e como matrizes de adjacências. A Seção 22.2 apresenta um algoritmo simples de busca em grafos, denominado busca em largura, e mostra como criar uma árvore de busca em largura. A Seção 22.3 apresenta a busca em profundidade e prova alguns resultados padrões para a ordem em que a busca em profundidade visita os vértices. A Seção 22.4 dá nossa primeira aplicação real de busca em profundidade: ordenação topológica em um grafo acíclico dirigido. Uma segunda aplicação da busca em profundidade, encontrar as componentes fortemente conexas de um grafo dirigido, é o tópico da Seção 22.5.

22.1 REPRESENTAÇÕES DE GRAFOS

Podemos escolher entre dois modos padrões para representar um grafo $G = (V, E)$: como uma coleção de listas de adjacências ou como uma matriz de adjacências. Qualquer desses modos se aplica a grafos dirigidos e não dirigidos. Como a representação por lista de adjacências nos dá um modo compacto de representar grafos ***esparsos*** — aqueles para os quais $|E|$ é muito menor que $|V|^2$ —, ela é, em geral, o método preferido. A maioria dos algoritmos de grafos apresentados neste livro supõe que um grafo de entrada é representado sob a forma de lista de adjacências. Contudo, uma representação por matriz de adjacências pode ser preferível quando o grafo é ***denso*** — $|E|$ está próximo de $|V|^2$ — ou quando precisamos saber rapidamente se há uma aresta conectando dois vértices dados. Por exemplo, dois dos algoritmos de caminhos mínimos para todos os pares apresentados no Capítulo 25 supõem que seus grafos de entrada são representados por matrizes de adjacências.

A ***representação por lista de adjacências*** de um grafo $G = (V, E)$ consiste em um arranjo Adj. de $|V|$ listas, uma para cada vértice em V. Para cada $u \in V$, a lista de adjacências $Adj[u]$ contém todos o vértices v tais que existe uma aresta $(u, v) \in E$. Isto é, $Adj[u]$ consiste em todos os vértices adjacentes a u em G. (Alternativamente, ela pode conter ponteiros para esses vértices.) Visto que as listas de adjacências representam os vértices de um grafo, em pseudocódigo tratamos o arranjo Adj como um atributo do grafo, exatamente como tratamos o conjunto de vértices E. Portanto, em pseudocódigo, veremos notação tal como G.$Adj[u]$. A Figura 22.1(b) é uma representação por lista de adjacências do grafo não dirigido na Figura 22.1(a). De modo semelhante, a Figura 22.2(b) é uma representação por lista de adjacências do grafo dirigido na Figura 22.2(a).

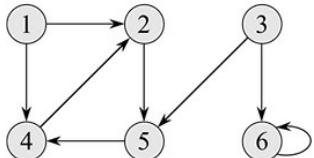
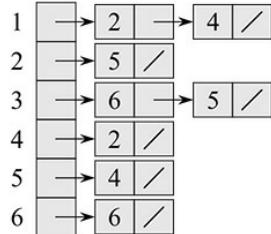



Figura 22.1 Duas representações de um grafo não dirigido. (**a**) Um grafo não dirigido *G* com cinco vértices e sete arestas. (**b**) Uma representação de *G* por lista de adjacências. (**c**) A representação de *G* por matriz de adjacências.

Figura 22.2 Duas representações de um grafo dirigido. (**a**) Um grafo dirigido *G* com seis vértices e oito arestas. (**b**) Uma representação de *G* por lista de adjacências. (**c**) A representação de *G* por matriz de adjacências.

Se G é um grafo dirigido, a soma dos comprimentos de todas as listas de adjacências é $|E|$, já que uma aresta da forma (u, v) é representada fazendo com que v apareça em $Adj[u]$. Se G é um grafo não dirigido, a soma dos comprimentos de todas as listas de adjacências é $2|E|$, já que, se (u, v) é uma aresta não dirigida, então u aparece na lista de adjacências de v e *vice-versa*. Quer os grafos sejam dirigidos ou não dirigidos, a representação por lista de adjacências tem a seguinte propriedade interessante: a quantidade de memória que ela exige é Q($V + E$).

Podemos adaptar imediatamente as listas de adjacências para representar ***grafos ponderados***, isto é, grafos nos quais cada aresta tem um ***peso*** associado, normalmente dado por uma ***função peso*** $w : E \rightarrow$. Por exemplo, seja $G = (V, E)$ um grafo ponderado com função peso w. Simplesmente armazenamos o peso $w(u, v)$ da aresta $(u, v) \in E$ com o vértice v na lista de adjacências de u. A representação por lista de adjacências é bastante robusta no sentido de que podemos modificá-la para suportar muitas outras variantes de grafos.

Uma desvantagem potencial da representação por lista de adjacências é que ela não proporciona nenhum modo mais rápido para determinar se uma dada aresta (u, v) está presente no grafo do que procurar v na lista de adjacências $Adj[u]$. Essa desvantagem pode ser contornada por uma representação por matriz de adjacências do grafo, porém ao custo de utilizar assintoticamente mais memória. (Veja no Exercício 22.1-8 sugestões de variações para listas de adjacências que permitem busca mais rápida de arestas.)

No caso da ***representação por matriz de adjacências*** de um grafo $G = (V, E)$, supomos que os vértices são numerados 1, 2, ..., $|V|$ de alguma maneira arbitrária. Então, a representação por matriz de adjacências de um grafo G consiste em uma matriz $|V| \times |V|$ $A = (a_{ij})$ tal que

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E, \\ 0 & \text{caso contrário} \end{cases}$$

As Figuras 22.1(c) e 22.2(c) são as matrizes de adjacências do grafo não dirigido e do grafo dirigido nas Figuras 22.1(a) e 22.2(a), respectivamente. A matriz de adjacências de um grafo exige Q(V_2) de memória, independentemente do número de arestas no grafo.

Observe a simetria ao longo da diagonal principal da matriz de adjacências na Figura 22.1(c). Visto que, em um gráfico não dirigido, (u, v) e (v, u) representam a mesma aresta, a matriz de adjacências A de um grafo não dirigido é sua própria transposta: $A = A_T$. Em algumas aplicações, vale a pena armazenar somente as entradas que estão na diagonal e acima da diagonal da matriz de adjacências, o que reduz quase à metade a memória necessária para armazenar o grafo.

Assim como a representação por lista de adjacências de um grafo, uma matriz de adjacências pode representar um grafo ponderado. Por exemplo, se $G = (V, E)$ é um grafo ponderado com função peso de aresta w, podemos simplesmente armazenar o peso $w(u, v)$ da aresta $(u, v) \in E$ como a entrada na linha u e coluna v da matriz de adjacências. Se uma aresta não existe, podemos armazenar um valor NIL como sua entrada de matriz correspondente, se bem que em muitos problemas é conveniente usar um valor como 0 ou ∞.

Embora a representação por lista de adjacências seja assintoticamente, no mínimo, tão eficiente em termos de espaço quanto a representação por matriz de adjacências, matrizes de adjacências são mais simples e portanto preferíveis quando os grafos são razoavelmente pequenos. Além disso, matrizes de adjacências têm uma vantagem adicional para grafos não ponderados: exigem somente um bit por entrada.

Representação de atributos

A maioria dos algoritmos que funcionam em grafos precisa manter atributos para vértices e/ou arestas. Indicamos esses atributos usando nossa notação usual, por exemplo, $v.d$ para um atributo d de um vértice v. Quando indicamos arestas como pares de vértices, usamos o mesmo estilo de notação. Por exemplo, se arestas têm um atributo f, denotamos esse atributo para a aresta (u, v) por $(u, v).f$. Para a finalidade de apresentar e entender algoritmos, nossa notação de atributo é suficiente.

Implementar atributos de vértice e aresta em programas reais pode ser uma história inteiramente diferente. Não há nenhum modo que seja reconhecidamente melhor para armazenar e acessar atributos de vértice e de aresta. Dada uma situação, é provável que sua decisão dependerá da linguagem de programação que estiver usando, do algoritmo que estiver implementando e de como o resto de seu programa usará o grafo. Se você representar um grafo utilizando listas de adjacências, um projeto possível representa atributos de vértice em arranjos adicionais, tal como um arranjo $d[1 \, . \, . \, |V|]$, que é paralelo ao arranjo Adj. Se os vértices adjacentes a u estão em $Adj[u]$, então aquilo que denominamos atributo $u.d$ seria realmente armazenado na entrada de arranjo $d[u]$. Há muitos outros modos possíveis de implementar atributos. Por exemplo, em uma linguagem de programação orientada a objeto, atributos de vértices poderiam ser representados como variáveis de instâncias dentro de uma subclasse de uma classe Vertex.

Exercícios

22.1-1 Dada uma representação por lista de adjacências de um grafo dirigido, qual o tempo necessário para calcular os graus de saída de todos os vértices? Qual o tempo necessário para calcular os graus de entrada?

22.1-2 Dê uma representação por lista de adjacências para uma árvore binária completa em sete vértices. Dê uma representação por matriz de adjacências equivalente. Suponha que os vértices são numerados de 1 até 7 como em um heap binário.

22.1-3 O ***transposto*** de um grafo dirigido $G = (V, E)$ é o grafo $G_T = (V, E_T)$, onde $E_T = \{(v, u) \in V \times V : (u, v) \in E\}$. Assim, G_T é G com todas as suas arestas invertidas. Descreva algoritmos eficientes para calcular G_T a partir de G, para a representação por lista de adjacências e também para a representação por matriz de adjacências de G. Analise os tempos de execução de seus algoritmos.

22.1-4 Dada uma representação por lista de adjacências de um multigrafo $G = (V, E)$, descreva um algoritmo de tempo $O(V + E)$ para calcular a representação por lista de adjacências do grafo não dirigido "equivalente" $G' = (V, E')$, onde E' consiste nas arestas em E onde todas as arestas múltiplas entre dois vértices foram substituídas por uma aresta única e onde todos os laços foram removidos.

22.1-5 O ***quadrado*** de um grafo dirigido $G = (V, E)$ é o grafo $G_2 = (V, E_2)$ em que $(u, v) \in E_2$ se e somente se G contiver um caminho que tenha no máximo duas arestas entre u e v. Descreva algoritmos eficientes para calcular G_2 a partir de G para uma representação por lista de adjacências e para uma representação por matriz de adjacências de G. Analise os tempos de execução de seus algoritmos.

22.1-6 A maioria dos algoritmos em grafos que adota uma representação por matriz de adjacências como entrada exige o tempo (V_2), mas há algumas exceções. Mostre como determinar se um grafo dirigido G contém um ***sorvedouro universal*** — um vértice com grau de entrada $|V| - 1$ e grau de saída 0 — no tempo $O(V)$, dada uma matriz de adjacências para G.

22.1-7 A ***matriz de incidência*** de um grafo dirigido $G = (V, E)$ sem nenhum laço é uma matriz $|V| \times |E|$ $B = (b_{ij})$ tal que

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i, \\ 1 & \text{if edge } j \text{ enters vertex } i, \\ 0 & \text{caso contrário.} \end{cases}$$

Descreva o que representam as entradas do produto de matrizes BB_T, onde B_T é a transposta de B.

22.1-8 Suponha que, em vez de uma lista ligada, cada entrada de arranjo $Adj[u]$ seja uma tabela de espalhamento que contém os vértices v para os quais $(u, v) \in E$. Se todas as buscas de arestas forem igualmente prováveis, qual é o tempo esperado para determinar se uma aresta está no grafo? Quais são as desvantagens desse esquema? Sugira uma estrutura de dados alternativa para cada lista de arestas que resolva esses problemas. Sua alternativa tem desvantagens em comparação com a tabela de espalhamento?

22.2 BUSCA EM LARGURA

A ***busca em largura*** é um dos algoritmos mais simples para executar busca em um grafo e é o arquétipo de muitos algoritmos de grafos importantes. O algoritmo de árvore geradora mínima de Prim (Seção 23.2) e o algoritmo de caminhos mínimos de fonte única de Dijkstra (Seção 24.3) usam ideias semelhantes às que aparecem na busca em largura.

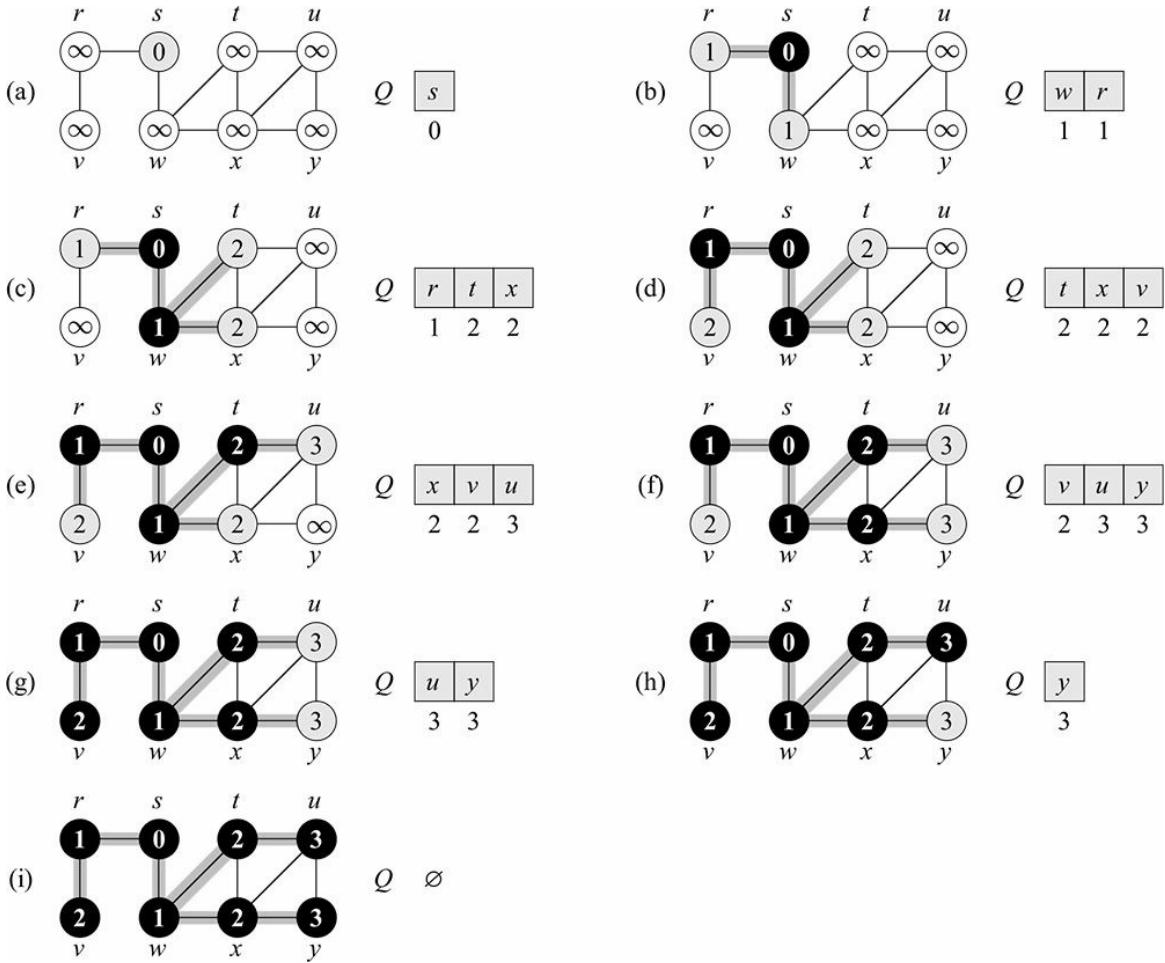
Dado um grafo $G = (V, E)$ e um vértice ***fonte*** s, a busca em largura explora sistematicamente as arestas de G para "descobrir" cada vértice que pode ser alcançado a partir de s. O algoritmo calcula a distância (menor número de arestas) de s até cada vértice que pode ser alcançado. Produz também uma "árvore de busca em largura" com raiz s que contém todos os vértices que podem ser alcançados. Para qualquer vértice v que pode ser alcançado de s, o caminho simples na árvore de busca em largura de s até v corresponde a um "caminho mínimo" de s a v em G, isto é, um caminho que contém o menor número de arestas. O algoritmo funciona em grafos dirigidos, bem como em grafos não dirigidos.

A busca em largura tem esse nome porque expande a fronteira entre vértices descobertos e não descobertos uniformemente ao longo da extensão da fronteira. Isto é, o algoritmo descobre todos os vértices à distância k de s, antes de descobrir quaisquer vértices à distância $k + 1$.

Para controlar o progresso, a busca em largura pinta cada vértice de branco, cinzento ou preto. No início, todos os vértices são brancos, e mais tarde eles podem se tornar cinzentos e depois pretos. Um vértice é ***descoberto*** na primeira

vez em que é encontrado durante a busca, e nesse momento ele se torna não branco. Portanto, vértices cinzentos e pretos são vértices descobertos, mas a busca em largura distingue entre eles para assegurar que a busca prossiga sendo em largura.[1] Se $(u, v) \in E$ e o vértice u é preto, então o vértice v é cinzento ou preto; isto é, todos os vértices adjacentes a vértices pretos foram descobertos. Vértices cinzentos podem ter alguns vértices adjacentes brancos; eles representam a fronteira entre vértices descobertos e não descobertos.

A busca em largura constrói uma árvore em largura, que contém inicialmente apenas sua raiz, que é o vértice de fonte s. Sempre que a busca descobre um vértice branco v no curso da varredura da lista de adjacências de um vértice u já descoberto, o vértice v e a aresta (u, v) são acrescentados à árvore. Dizemos que u é o ***predecessor*** ou ***pai*** de v na árvore de busca em largura. Visto que um vértice é descoberto no máximo uma vez, ele tem no máximo um pai. Relações de ancestral e descendente na árvore de busca em largura são definidas em relação à raiz s da maneira usual: se u está em um caminho simples na árvore que vai da raiz s até o vértice v, então u é um ancestral de v,e v é um descendente de u.

O procedimento de busca em largura BFS mostrado a seguir supõe que o grafo de entrada $G = (V, E)$ é representado com a utilização de listas de adjacências. Ele anexa vários atributos adicionais a cada vértice no grafo. Armazenamos a cor de cada vértice $u \in V$ no atributo $u.cor$ e o predecessor de u no atributo $u.p$. Se u não tem nenhum predecessor (por exemplo, se $u = s$ ou se u não foi descoberto), então $u.p$ = NIL. O atributo $u.d$ mantém a distância da fonte s ao vértice u calculada pelo algoritmo. O algoritmo também utiliza uma fila Q do tipo primeiro a entrar, primeiro a sair (veja a Seção 10.1) para gerenciar o conjunto de vértices cinzentos.

```
BFS(G, s)
 1   for cada vértice u ∈ V[G] − {s}
 2       u.cor = BRANCO
 3       u.d = ∞
 4       u.π = NIL
 5   s.cor = CINZENTO
 6   s.d = 0
 7   s.π = NIL
 8   Q = ∅
 9   ENQUEUE(Q, s)
10   while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for cada v = Adj[u]
13           if v.cor == BRANCO
14               v.cor == CINZENTO
15               v.d = u.d + 1
16               v.π = u
17               ENQUEUE(Q, v)
18       u.cor = PRETO
```

A Figura 22.3 ilustra o progresso de BFS em um grafo exemplo.

O procedimento BFS funciona da maneira descrita a seguir. Com a exceção do vértice de fonte s, *as* linhas 1–4 pintam todos os vértices de branco, definem $u.d$ como infinito para todo vértice u e definem o pai de todo vértice como NIL. A linha 5 pinta s de cinzento, já que consideramos que ele é descoberto quando o procedimento começa. A linha 6 inicializa $s.d$ como 0, e a linha 7 define o predecessor do fonte como NIL. As linhas 8–9 inicializam Q como a fila que contém apenas o vértice s.

Figura 22.3 O funcionamento de BFS em um grafo não dirigido. As arestas da árvore aparecem sombreadas à medida que são produzidas por BFS. O valor de $u.d$ aparece dentro de cada vértice u. A fila Q é mostrada no início de cada iteração do laço **while** das linhas 10-18. As distâncias de vértices são mostradas abaixo dos vértices na fila.

O laço **while** das linhas 10–18 itera enquanto houver vértices cinzentos, que são vértices descobertos cujas listas de adjacências ainda não foram totalmente examinadas. Esse laço **while** mantém o seguinte invariante:

No teste na linha 10, a fila Q consiste no conjunto de vértices cinzentos.

Se bem que não usaremos esse invariante de laço para provar correção, é fácil ver que ele é válido antes da primeira iteração e que cada iteração do laço mantém o invariante. Antes da primeira iteração, o único vértice cinzento, e o único vértice em Q, é o vértice de fonte s. A linha 11 determina o vértice cinzento u no início da fila Q e o remove de Q. O laço **for** das linhas 12–17 considera cada vértice v na lista de adjacências de u. Se v é branco, então ainda não foi descoberto, e o procedimento o descobre executando as linhas 14–17. O procedimento pinta o vértice v de cinzento, define sua distância $v.d$ como $u.d + 1$, registra u como seu pai $v.p$ e o coloca no final da fila Q. Uma vez examinados todos os vértices na lista de adjacências de u, o procedimento pinta u de preto na linha 18. O invariante de laço é mantido porque sempre que um vértice é pintado de cinzento (na linha 14), ele é também enfileirado (na linha 17) e, sempre que um vértice é retirado da fila (na linha 11), ele é também pintado de preto (na linha 18).

Os resultados da busca em largura podem depender da ordem na qual os vizinhos de um determinado vértice são visitados na linha 12; a árvore de busca em largura pode variar, mas as distâncias d calculadas pelo algoritmo não variam (veja o Exercício 22.2-5).

Análise

Antes de provar as várias propriedades da busca em largura, realizamos o trabalho um pouco mais fácil de analisar seu tempo de execução em um grafo de entrada $G = (V, E)$. Utilizamos análise agregada, como vimos na Seção 17.1. Após a inicialização, a busca em largura nunca pinta um vértice de branco e, assim, o teste na linha 13 assegura que cada vértice seja colocado na fila no máximo uma vez, e portanto, é retirado da fila no máximo uma vez. As operações de enfileirar e desenfileirar demoram o tempo $O(1)$ e, assim, o tempo total dedicado a operações de fila é $O(V)$. Como o procedimento varre a lista de adjacências de cada vértice somente quando o vértice é desenfileirado, varre cada linha de adjacências no máximo uma vez. Visto que a soma dos comprimentos de todas as listas de adjacências é Q(E), o tempo total gasto na varredura das listas de adjacências é $O(E)$. A sobrecarga de inicialização é $O(V)$ e, portanto, o tempo de execução total do procedimento BFS é $O(V + E)$. Assim, a busca em largura é executada em tempo linear em relação ao tamanho da representação por lista de adjacências de G.

Caminhos mínimos

No início desta seção, afirmamos que a busca em largura encontra a distância até cada vértice que pode ser alcançado em um grafo $G = (V, E)$ partindo de um determinado vértice de fonte $s \in V$. Defina a ***distância do caminho mínimo*** $d(s, v)$ de s a v como o número mínimo de arestas em qualquer caminho do vértice s ao vértice v; se não há nenhum caminho de s a v, então $d(s, v) = \infty$. Denominamos um caminho de comprimento $d(s, v)$ de s a v ***caminho mínimo***[2] de s a v. Antes de mostrar que a busca em largura calcula corretamente distâncias de caminhos mínimos, examinamos uma propriedade importante de distâncias de caminhos mínimos.

Lema 22.1

Seja $G = (V, E)$ um grafo dirigido ou não dirigido e seja $s \in V$ um vértice arbitrário. Então, para qualquer aresta $(u, v) \in E$, >$d(s, v) \leq d(s, u) + 1$.

Prova Se u pode ser alcançado a partir de s, então o mesmo ocorre com v. Nesse caso, o caminho mínimo de s a v não pode ser mais longo que o caminho mínimo de s a u seguido pela aresta (u, v) e. assim. a desigualdade vale. Se u não pode ser alcançado de s, então $d(s, u) = \infty$, e a desigualdade é válida.

Queremos mostrar que BFS calcula adequadamente $v.d = d(s, v)$ para cada vértice $v \in V$. Primeiro, mostramos que $v.d$ limita $d(s, v)$ por cima.

Lema 22.2

Seja $G = (V, E)$ um grafo dirigido ou não dirigido e suponha que BFS seja executado em G partindo de um dado vértice de fonte $s \in V$. Então, no término, para cada vértice $v \in V$, o valor $v.d$ calculado por BFS satisfaz $v.d \geq d(s, v)$.

Prova Utilizamos indução em relação ao número de operações ENQUEUE. Nossa hipótese indutiva é que $v.d \geq$ d(s, v) para todo $v \in V$.
A base da indução é a situação imediatamente após s ser enfileirado na linha 9 de BFS. Aqui, a hipótese indutiva se mantém válida porque $s.d = 0 = d(s, s)$ e $v.d = \infty \geq d(s, s)$ para todo $v \in V - \{s\}$.

Para o passo de indução, considere um vértice branco v que é descoberto durante a busca de um vértice u. A hipótese de indução implica que $u.d \geq d(s, u)$. Pela atribuição executada pela linha 15 e pelo Lema 22.1, obtemos

$$\begin{aligned} v.d &= u.d + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v)\ . \end{aligned}$$

Então o vértice v é enfileirado e nunca será enfileirado novamente porque ele também é pintado de cinzento, e a cláusula **then** das linhas 14–17 é executada somente para vértices brancos. Assim, o valor de $v.d$ nunca muda novamente, e a hipótese de indução é mantida.

Para provar que $d[v] = d(s, v)$, primeiro devemos mostrar com mais precisão como a fila Q funciona durante o curso de BFS. O próximo lema mostra que, em todas as vezes, a fila contém no máximo dois valores d distintos.

Lema 22.3

Suponha que durante a execução de BFS em um grafo $G = (V, E)$, a fila Q contenha os vértices $\langle v_1, v_2, ..., v_r \rangle$, onde v_1 é o início de Q e v_r é o final. Então, $v_r.d \leq v_1.d + 1$ e $v_i.d \leq v_{i+1}.d$ para $i = 1, 2, ..., r - 1$.

Prova A prova é por indução em relação ao número de operações de fila. Inicialmente, quando a fila contém apenas s, o lema certamente é válido.

Para o passo de indução, devemos provar que o lema se mantém válido tanto depois do desenfileiramento quanto do enfileiramento de um vértice. Se o início v_1 da fila é desenfileirado, v_2 torna-se o novo início. (Se a fila se torna vazia, então o lema se mantém válido vacuosamente.)

Pela hipótese de indução, $v_1.d \leq v_2.d$. Mas, então, temos $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$, e as desigualdades restantes não são afetadas. Assim, o lema prossegue com v_2 como início. Para entender o que acontece quando enfileiramos um vértice, precisamos examinar o código mais minuciosamente. Quando enfileiramos um vértice v na linha 17 de BFS, ele se torna v_{r+1}. Nesse momento, já removemos da fila Q o vértice u cuja lista de adjacências está sendo examinada e, pela hipótese de indução, o novo início v_1 tem $v_1.d \geq u.d$. Assim, $v_{i+1}.d = v.d = u.d + 1 \leq v_1.d + 1$. Pela hipótese indutiva, temos também $v_r.d \leq u.d + 1$ e, portanto, $v_r.d \leq u.d + 1 = v.d = v_{i+1}.d$, e as desigualdades restantes não são afetadas. Portanto, o lema prossegue quando v é enfileirado.

O corolário a seguir mostra que os valores d no momento em que os vértices são enfileirados são monotonicamente crescentes com o tempo.

Corolário 22.4

Suponha que os vértices v_i e v_j sejam enfileirados durante a execução de BFS e que v_i seja enfileirado antes de v_j. Então, $v_i.d \leq v_j.d$ no momento em que v_j é enfileirado.

Prova Imediata pelo Lema 22.3 e pela propriedade de que cada vértice recebe um valor d finito no máximo uma vez durante a execução de BFS.

Agora, podemos provar que a busca em largura encontra corretamente distâncias de caminhos mínimos.

Teorema 22.5 (Correção da busca em largura)

Seja $G = (V, E)$ um grafo dirigido ou não dirigido, e suponha que BFS seja executado em G partindo de um dado vértice de fonte $s \in V$. Então, durante sua execução, BFS descobre todo vértice $v \in V$ que pode ser alcançado da fonte s e, no término, $v.d = d(s, v)$ para todo $v \in V$. Além disso, para qualquer vértice $v \neq s$ que pode ser alcançado de s, um dos caminhos mínimos de s a v é um caminho mínimo de s a $v.p$ seguido pela aresta $(v.p, v)$.

Prova Suponha, por contradição, que algum vértice receba um valor d não igual à distância de seu caminho mínimo. Seja v o vértice com $d(s, v)$ mínimo que recebe tal valor d incorreto; é claro que $v \neq s$. Pelo Lema 22.2, $v.d \geq d(s, v)$ e, portanto, temos que $v.d > d(s, v)$. O vértice v deve poder ser visitado de s porque, se não puder, $d(s, v) = \infty \geq v.d$.

Seja u o vértice imediatamente anterior a v em um caminho mínimo de s a v, de modo que $d(s, v) = d(s, u) + 1$. Como $d(s, u) < d(s, v)$, e em razão do modo como escolhemos v, temos $u.d = d(s, u)$. Reunindo essas propriedades, temos

$$v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1 . \tag{22.1}$$

Agora considere o momento em que BFS opta por desenfileirar o vértice u de Q na linha 11. Nesse momento, o vértice v é branco, cinzento ou preto. Mostraremos que, em cada um desses casos, deduzimos uma contradição para a desigualdade (22.1). Se v é branco, então a linha 15 define $v.d = u.d + 1$, contradizendo a desigualdade (22.1). Se v é preto, então já foi removido da fila e, pelo Corolário 22.4, temos $v.d \leq u.d$, que novamente contradiz a desigualdade (22.1). Se v é cinzento, então ele foi pintado de cinzento ao ser desenfileirado algum vértice w que foi removido de Q antes de u e para o qual $v.d = w.d + 1$. Porém, pelo Corolário 22.4, $w.d \leq u.d$ e, então, temos $v.d = w.d + 1 \leq u.d + 1$, uma vez mais contradizendo a desigualdade (22.1). Assim, concluímos que $v.d = d(s, v)$ para todo $v \in V$. Todos os vértices que podem ser visitados de s devem ser descobertos porque, se não fossem, teriam $\infty = v.d > d(s, v)$. Para concluir a prova do teorema, observe que, se $v.p = u$, então $v.d = u.d + 1$. Assim, podemos obter um caminho mínimo de s a v tomando um caminho mínimo de s a $v.p$ e depois percorrendo a aresta $(v.p, v)$.

Árvores em largura

O procedimento BFS constrói uma árvore de busca em largura à medida que efetua a busca no grafo, como ilustra a Figura 22.3. A árvore corresponde aos atributos p. Em linguagem mais formal, para um grafo $G = (V, E)$ com fonte s, definimos o ***subgrafo dos predecessores*** de G como $G_p = (V_p, E_p)$, onde

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

e

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\} .$$

O subgrafo dos predecessores G_p é uma ***árvore de busca em largura*** se V_p consistir nos vértices que podem ser visitados de s e, para todo $v \in V_p$, existe um caminho simples único de s a v em G_p que também é um caminho mínimo de s a v em G. Uma árvore de busca em largura é, na verdade, uma árvore, já que é conexa e $|E_p| = |V_p| - 1$ (ver Teorema B.2). As arestas em E_p são denominadas ***arestas da árvore***.

O lema a seguir mostra que o subgrafo dos predecessores produzido pelo procedimento BFS é uma árvore de busca em largura.

Lema 22.6

Quando aplicado a um grafo dirigido ou não dirigido $G = (V, E)$, o procedimento BFS constrói p de tal forma que o subgrafo predecessor $G_p = (V_p, E_p)$ é uma busca árvore em largura.

Prova A linha 16 de BFS define $v.p = u$ se e somente se $(u, v) \in E$ e $d(s, v) < \infty$ — isto é, se v pode ser visitado por s — e, assim, V_p consiste nos vértices em V que podem ser visitados por s. Visto que G_p forma uma árvore, pelo Teorema B.2 ela contém um caminho simples único de s a cada vértice em V_p. Aplicando o Teorema 22.5 por indução, concluímos que todo caminho desse tipo é um caminho mínimo em G.

O seguinte procedimento imprime os vértices em um caminho mínimo de s a v, considerando que BFS já tenha calculado uma árvore em largura:

```
PRINT-PATH(G, s, v)
1   if v == s
2         imprimir s
3   else if v.π = NIL
4         imprimir "não existe nenhum caminho de" s "para v"
5   else PRINT-PATH(G, s, v.π)
6         imprimir v
```

Esse procedimento é executado em tempo linear em relação ao número de vértices no caminho impresso, já que cada chamada recursiva é para um caminho um vértice mais curto.

Exercícios

22.2-1 Mostre os valores de d e p que resultam da execução da busca em largura no grafo dirigido da Figura 22.2(a), usando o vértice 3 como fonte.

22.2-2 Mostre os valores d e p que resultam da execução da busca em largura no grafo não dirigido da Figura 22.3, usando o vértice u como fonte.

22.2-3 Mostre que usar um único bit para armazenar cada cor de vértice é suficiente demonstrando que o procedimento BFS produziria o mesmo resultado se a linha 18 fosse removida.

22.2-4 Qual é o tempo de execução de BFS se representarmos o seu grafo de entrada por uma matriz de adjacências e modificarmos o algoritmo para tratar essa forma de entrada?

22.2-5 Mostre que, em uma busca em largura, o valor $u.d$ atribuído a um vértice u é independente da ordem na qual os vértices aparecem em cada lista de adjacências. Usando a Figura 22.3 como exemplo, mostre que a árvore de busca em largura calculada por BFS pode depender da ordenação dentro de listas de adjacências.

22.2-6 Dê um exemplo de grafo dirigido $G = (V, E)$, um vértice fonte $s \in V$ e um conjunto de arestas de árvore $E_p \subseteq E$ tal que, para cada vértice $v \in V$, o caminho simples único no grafo (V, E_p) de s a v é um caminho mínimo em G e que, ainda assim, o conjunto de arestas E_p não pode ser produzido executando-se BFS em G, não importando como os vértices estão ordenados em cada lista de adjacências.

22.2-7 Há dois tipos de lutadores profissionais: os "bonzinhos" e os "vilões". Entre qualquer par de lutadores profissionais pode ou não haver uma rivalidade. Suponha que tenhamos n lutadores profissionais e uma lista de r pares de lutadores entre os quais há rivalidade. Dê um algoritmo de tempo $O(n + r)$ que determine se é possível designar alguns dos lutadores como bonzinhos e os restantes como vilões, de modo que a rivalidade ocorra sempre entre um bonzinho e um vilão. Se for possível realizar tal designação, seu algoritmo deve produzi-la.

22.2-8 ★ O ***diâmetro*** de uma árvore $T = (V, E)$ é definido por $\max_{u,v \in V} d(u, v)$ isto é, a maior de todas distâncias de caminhos mínimos na árvore. Dê um algoritmo eficiente para calcular o diâmetro de uma árvore e analise o tempo de execução de seu algoritmo.

22.2-9 Seja $G = (V, E)$ um grafo conexo não dirigido. Dê um algoritmo de tempo $O(V + E)$ para calcular um caminho em G que percorra cada aresta em E exatamente uma vez em cada direção. Descreva como você pode encontrar a saída de um labirinto se dispuser de uma grande provisão de moedas de um centavo.

22.3 BUSCA EM PROFUNDIDADE

A estratégia seguida pela busca em profundidade é, como seu nome implica, buscar "mais fundo" no grafo, sempre que possível. A busca em profundidade explora arestas partindo do vértice v mais recentemente descoberto do qual ainda saem arestas inexploradas. Depois que todas as arestas de v foram exploradas, a busca "regressa pelo mesmo caminho" para explorar as arestas que partem do vértice do qual v foi descoberto. Esse processo continua até descobrirmos todos os vértices que podem ser visitados a partir do vértice fonte inicial. Se restarem quaisquer vértices não descobertos, a busca em profundidade seleciona um deles como fonte e repete a busca partindo dessa fonte. O algoritmo repete esse processo inteiro até descobrir todos os vértices.[3]

Como ocorre na busca em largura, sempre que a busca em profundidade descobre um vértice v durante uma varredura da lista de adjacências de um vértice já descoberto u, registra esse evento definindo o atributo predecessor de v, $v.p$ como u. Diferentemente da busca em largura, cujo subgrafo dos predecessores forma uma árvore, o subgrafo dos predecessores produzido por uma busca em profundidade pode ser composto por várias árvores porque a busca pode ser repetida partindo de várias fontes. Portanto, definimos o ***subgrafo dos predecessores*** de uma busca em profundidade de um modo ligeiramente diferente do da busca em largura: fazemos $G_p = (V, E_p)$, onde

$$E_\pi = \{(v.\pi, v) : v \in V \text{ e } v.\pi \neq \text{NIL}\} .$$

O subgrafo dos predecessores de uma busca em profundidade forma uma ***floresta de busca em profundidade*** que abrange várias ***árvores de busca em profundidade***. As arestas em E_p são ***arestas de árvore***.

Como na busca em largura, a busca em profundidade pinta os vértices durante a busca para indicar o estado de cada um. Cada vértice é inicialmente branco, pintado de cinzento quando ***descoberto*** na busca e pintado de preto quando ***terminado***, isto é, quando sua lista de adjacências já foi totalmente examinada. Essa técnica garante que cada vértice acabe em exatamente uma árvore, de forma que essas árvores são disjuntas.

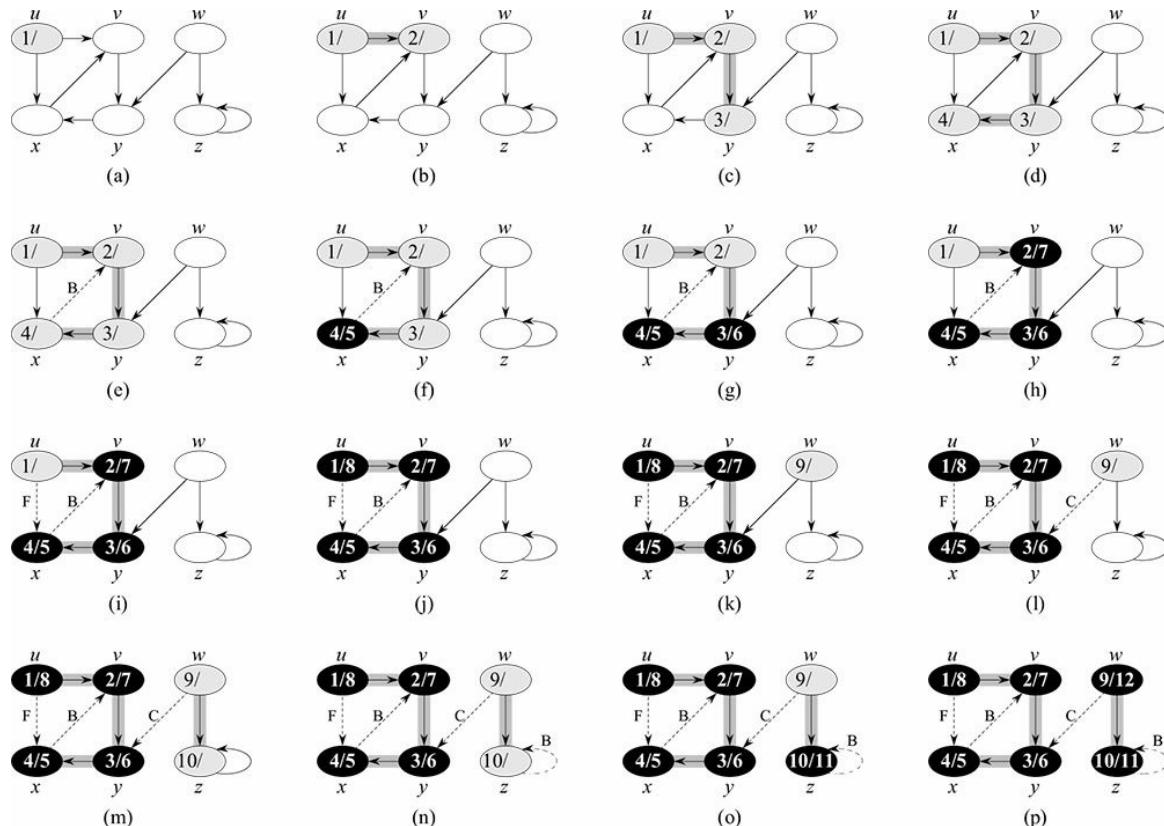
Além de criar uma floresta, a busca em profundidade também identifica cada vértice com um ***carimbo de tempo***. Cada vértice v tem dois carimbos de tempo: o primeiro carimbo de tempo $v.d$ registra quando v é descoberto pela primeira vez (e pintado de cinzento), e o segundo carimbo de tempo $v.f$ registra quando a busca termina de examinar a lista de adjacências de v (e pinta v de preto). Esses carimbos de tempo dão informações importantes sobre a estrutura do grafo e em geral são úteis para deduzir o comportamento da busca em profundidade.

O procedimento DFS a seguir registra no atributo $u.d$ o momento em que descobre o vértice u e registra no atributo $u.f$ o momento em que liquida o vértice u. Esses carimbos de tempo são inteiros entre 1 e 2 $|V|$, já que existe um evento de descoberta e um evento de término para cada um dos $|V|$ vértices. Para todo vértice u,

$$u.d < u.f . \tag{22.2}$$

O vértice u é BRANCO antes do tempo $u.d$, CINZENTO entre o tempo $u.d$ e o tempo $u.f$ e PRETO daí em diante.

O pseudocódigo a seguir é o algoritmo básico de busca em profundidade. O grafo de entrada G pode ser dirigido ou não dirigido. A variável *tempo* é uma variável global que utilizamos para definir carimbos de tempo.

```
DFS(G)
1       for cada vértice u ∈ V[G]
2               u.cor = BRANCO
3               u.π = NIL
4       tempo = 0
5       for cada vértice u ∈ V[G]
6               if u.cor == BRANCO
7                       DFS-VISIT(G, u)

DFS-VISIT(G, u)
1       tempo = tempo + 1                 // vértice branco u acabou de ser descoberto
2       u.d = tempo
3       u.cor = CINZENTO
4       for cada v ∈ G.Adj[u]             // explorar aresta (u, v)
5               if v.cor == BRANCO
6                       v.π = u
7                       DFS-VISIT(G, v)
8       u.cor = PRETO                     // pintar u de preto; está terminado
9       tempo = tempo + 1
10      u.f = tempo
```

A Figura 22.4 ilustra o progresso de DFS no grafo mostrado na Figura 22.2.

O procedimento DFS funciona da maneira descrita a seguir. As linhas 1–3 pintam todos os vértices de branco e inicializam seus atributos p como NIL. A linha 4 reajusta o contador de tempo global. As linhas 5–7 verificam cada vértice de V por vez e, quando um vértice branco é encontrado, elas o visitam usando DFS-VISIT. Toda vez que DFS-VISIT(G, u) é chamado na linha 7, o vértice u se torna a raiz de uma nova árvore na floresta em profundidade. Quando DFS retorna, a todo vértice u foi atribuído um ***tempo de descoberta*** $d[u]$ e um ***tempo de término*** $f[u]$.

Figura 22.4 Progresso do algoritmo de busca em profundidade DFS em um grafo dirigido. À medida que as arestas são exploradas pelo algoritmo, elas aparecem sombreadas (se são arestas de árvores) ou tracejadas (caso contrário). Arestas que não são de árvores são identificadas por B, C ou F, conforme sejam arestas de retorno, cruzadas ou diretas. Os carimbos de tempo dentro dos vértices indicam tempo de descoberta/tempo de término.

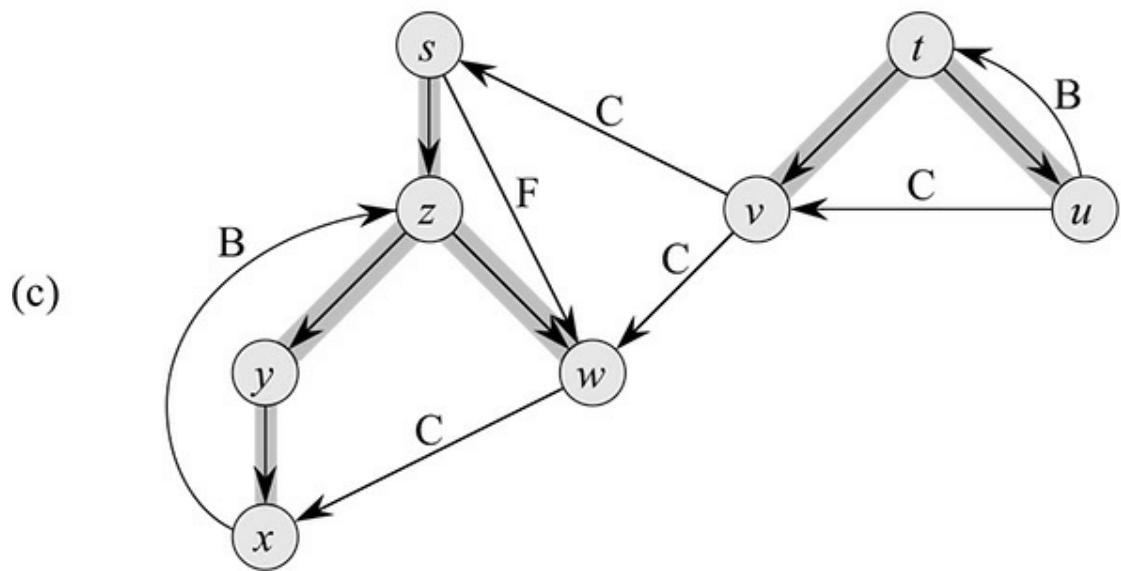
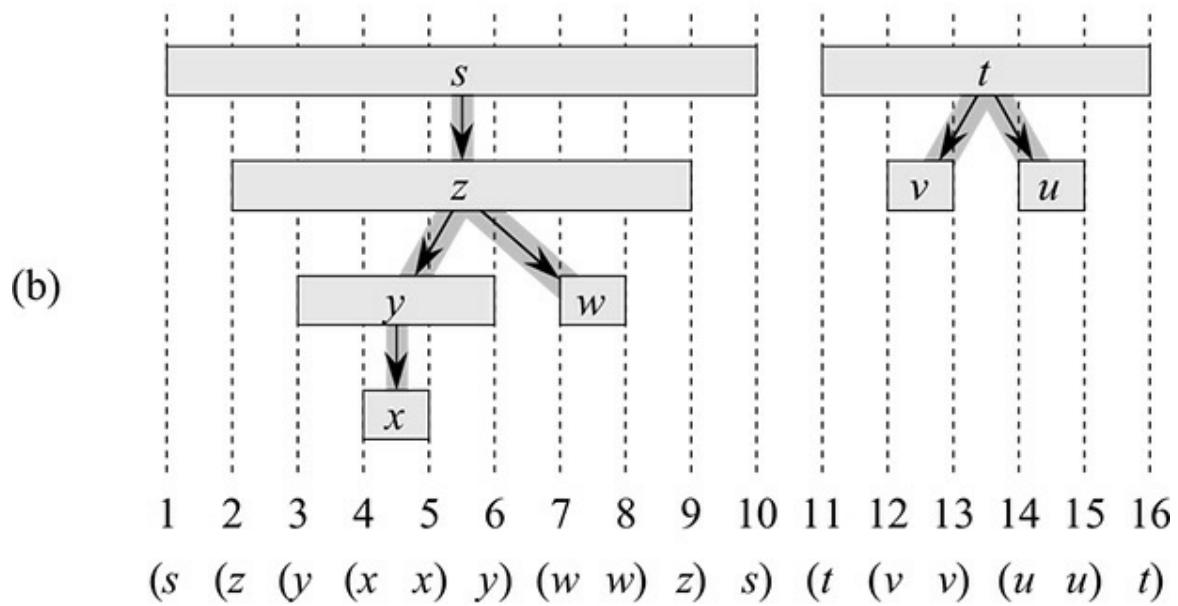
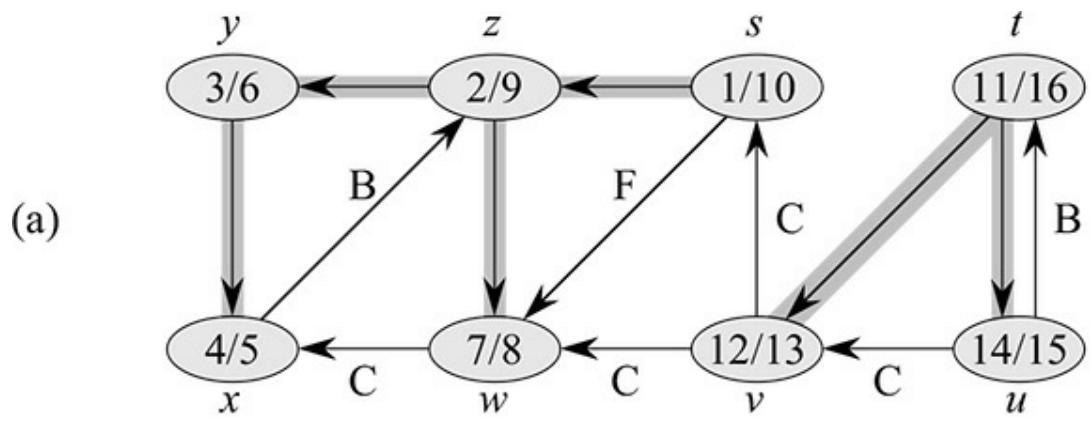
Em cada chamada DFS-Visit(G, u) o vértice u é inicialmente branco. A linha 1 incrementa a variável global *tempo*, a linha 2 registra o novo valor de *tempo* como o tempo de descoberta $d[u]$ e a linha 3 pinta u de cinzento. As linhas 4–7 examinam cada vértice v adjacente a u e visitam recursivamente v se ele é branco. À medida que cada vértice $v \in Adj[u]$ é considerado na linha 4, dizemos que a aresta (u, v) é ***explorada*** pela busca em profundidade. Finalmente, depois que toda aresta que sai de u foi explorada, as linhas 8–10 pintam u de preto, incrementam *tempo* e registram o tempo de término em $f[u]$.

Observe que os resultados da busca em profundidade podem depender da ordem em que a linha 5 de DFS examina os vértices e da ordem em que a linha 4 de DFS-Visit visita os vizinhos de um vértice. Essas diferentes ordens de visitação tendem a não causar problemas na prática, já que em geral podemos usar eficientemente *qualquer* resultado da busca em profundidade e obter, em essência, resultados equivalentes.

Qual é o tempo de execução de DFS? Os laços nas linhas 1–3 e nas linhas 5–7 de DFS demoram o tempo Q(V), excluindo o tempo para executar as chamadas a DFS-Visit. Como fizemos para a busca em largura, usamos análise agregada. O procedimento DFS-Visit é chamado exatamente uma vez para cada vértice $v \in V$, já que o vértice u no qual DFS-Visit é invocado tem de ser branco e a primeira coisa que DFS-Visit faz é pintar o vértice u de cinzento. Durante uma execução de DFS-Visit(Gv), o laço nas linhas 4–7 é executado $|Adj[v]|$ vezes. Visto que

$$\sum_{v \in V} |Adj[v]| = \Theta(E),$$

o custo total de executar as linhas 4–7 de DFS-Visit é Q(E). Portanto, o tempo de execução de DFS é Q($V + E$).

Propriedades da busca em profundidade

A busca em profundidade produz informações valiosas sobre a estrutura de um grafo. Talvez a propriedade mais básica da busca em profundidade seja que o subgrafo predecessor G_p realmente forma uma floresta de árvores, já que a estrutura das árvores em profundidade reflete exatamente a estrutura de chamadas recursivas de DFS-Visit. Isto é, $u = v.p$ se e somente se DFS-Visit(G, v) foi chamado durante uma busca da lista de adjacências de u. Além disso, o vértice v é um descendente do vértice u na floresta em profundidade se e somente se v é descoberto durante o tempo em que u é cinzento.

Uma outra propriedade importante da busca em profundidade é que os tempos de descoberta e término têm ***estrutura parentizada***. Se representarmos a descoberta do vértice u com um parêntese à esquerda "(u" e representarmos seu término por um parêntese à direita "u)", então a história de descobertas e términos gera uma expressão bem formada, no sentido de que os parênteses estão adequadamente aninhados. Por exemplo, a busca em profundidade da Figura 22.5(a) corresponde à parentização mostrada na Figura 22.5(b). O teorema a seguir dá um outro modo de caracterizar a estrutura parentizada.

Teorema 22.7 (Teorema dos parênteses)

Em qualquer busca em profundidade de um grafo (dirigido ou não dirigido) $G = (V, E)$, para quaisquer dois vértices u e v, exatamente uma das três condições seguintes é válida:

- Os intervalos $u.d$, $u.f$ e $v.d$, $v.f$ são completamente disjuntos, e nem u nem v é um descendente do outro na floresta em profundidade.

(a)
y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u

(b)
s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

(c)
s
t
C
B
z
F
v
C
u
B
C
y
w
C
x

Figura 22.5 Propriedades da busca em profundidade. (**a**) O resultado de uma busca em profundidade de um grafo dirigido. Os vértices são identificados por carimbos de tempo e os tipos de arestas são indicados como na Figura 22.4. (**b**) Os intervalos para o tempo de descoberta e o tempo de término de cada vértice correspondem à parentização mostrada. Cada retângulo compreende o intervalo dado pelos tempos de descoberta e término do vértice correspondente. Somente arestas de árvore são mostradas. Se dois intervalos se sobrepõem, então um deles está aninhado no outro, e o vértice correspondente ao menor intervalo é um descendente do vértice correspondente ao maior. (**c**) O grafo da parte (a) redesenhado com todas as arestas de árvore e diretas descendo no interior de uma árvore em profundidade e todas as arestas de retorno subindo de um descendente para um ancestral.

- O intervalo $u.d$, $u.f$ está contido inteiramente dentro do intervalo $v.d$, $v.f$, e u é um descendente de v em uma árvore em profundidade.
- O intervalo $v.d$, $v.f$ está contido inteiramente dentro do intervalo $u.d$, $u.f$, e v é um descendente de u em uma árvore em profundidade.

Prova Começamos com o caso no qual $u.d < v.d$. Consideramos dois subcasos, conforme $v.d < u.f$ ou não. O primeiro subcaso ocorre quando $v.d < u.f$, portanto v foi descoberto enquanto u ainda era cinzento, o que implica que v é um descendente de u. Além disso, como v foi descoberto mais recentemente que u, todas as suas arestas de saída são exploradas, e v é terminado antes de a busca retornar a u e terminá-lo. Portanto, nesse caso, o intervalo $[v.d, v.f]$ está completamente contido no intervalo $[u.d, u.f]$. No outro subcaso, $u.f < v.d$ e, pela desigualdade (22.2) $u.d < u.f < v.d < v.f$; assim, os intervalos $[u.d, u.f]$ e $[v.d, v.f]$ são disjuntos. Como os intervalos são disjuntos, nenhum dos vértices foi descoberto enquanto o outro era cinzento e, portanto, nenhum dos vértices é descendente do outro.
O caso em $v.d < u.d$ é semelhante, com os papéis de u e v invertidos no argumento anterior.

Corolário 22.8 (Aninhamento de intervalos de descendentes)

O vértice v é um descendente adequado do vértice u na floresta em profundidade para um grafo (dirigido ou não dirigido) G se e somente se $u.d < v.d < v.f < u.f$.

Prova Imediata, pelo Teorema 22.7.

O próximo teorema dá uma outra caracterização importante para quando um vértice é descendente de outro na floresta em profundidade.

Teorema 22.9 (Teorema do caminho branco)

Em uma floresta em profundidade de um grafo (dirigido ou não dirigido) $G = (V, E)$, o vértice v é um descendente do vértice u se e somente se no momento $u.d$ em que a busca descobre u, há um caminho de u a v que consiste inteiramente em vértices brancos.
Prova $\Rightarrow$: Se $v = u$, então o caminho de u a v contém apenas o vértice u, que ainda é branco quando definimos o valor de $u.d$. Agora, suponha que v seja um descendente próprio de u na floresta em profundidade. Pelo Corolário 22.8, $u.d < v.d$, *portanto*, v é branco no tempo $u.d$. Visto que v pode ser qualquer descendente de u, todos os vértices em um caminho simples único de u a v na floresta em profundidade são brancos no tempo $u.d$.
$\Leftarrow$: Suponha que haja um caminho de vértices brancos de u a v no tempo $u.d$, mas v não se torna um descendente de u na árvore em profundidade. Sem prejuízo da generalidade, considere que todo vértice exceto v ao longo do caminho se torne um descendente de u. (Caso contrário, seja v o vértice mais próximo de u ao longo do caminho que não se torna um descendente de u.) Seja w o predecessor de v no caminho, de modo que w seja um descendente de u (na verdade, w e u podem ser o mesmo vértice). Pelo Corolário 22.8, $w.f \leq u.f$. Como v tem de ser descoberto depois de u ser descoberto, mas antes de w ser terminado, temos $u.d < v.d < w.f \leq u.f$. Então, o Teorema 22.7 implica que o intervalo $[v.d, v.f]$ está contido inteiramente no intervalo $[u.d, u.f\,]$. Pelo Corolário 22.8, v deve ser, afinal, um descendente de u.

Classificação de arestas

Uma outra propriedade interessante da busca em profundidade é que a busca pode ser usada para classificar as arestas do grafo de entrada $G = (V, E)$. O tipo de cada aresta pode nos dar informações importantes sobre um grafo. Por exemplo, na próxima seção, veremos que um grafo dirigido é acíclico se e somente se uma busca em profundidade não produz nenhuma aresta "de retorno" (Lema 22.11).

Podemos definir quatro tipos de arestas em termos da floresta em profundidade G_p produzida por uma busca em profundidade em G:

1. ***Arestas de árvore*** são arestas na floresta em profundidade G_p. A aresta (u, v) é uma aresta de árvore se v foi descoberto primeiro pela exploração da aresta (u, v).
2. ***Arestas de retorno*** são as arestas (u, v) que conectam um vértice u a um ancestral v em uma árvore em profundidade. Consideramos laços, que podem ocorrer em grafos dirigidos, como arestas de retorno.
4. ***Arestas diretas*** são as arestas (u, v) não de árvore que conectam um vértice u a um descendente v em uma árvore em profundidade.
5. ***Arestas cruzadas*** são todas as outras arestas. Elas podem estar entre vértices na mesma árvore, desde que um vértice não seja um ancestral do outro, ou podem estar entre vértices em diferentes árvores de busca.

Nas Figuras 22.4 e 22.5, rótulos de arestas indicam os tipos de arestas. A Figura 22.5(c) também mostra como redesenhar o grafo da Figura 22.5(a) de modo que todas as arestas de árvore e diretas dirijam-se para baixo em uma árvore em profundidade e que todas as arestas de retorno dirijam-se para cima. Podemos redesenhar qualquer grafo dessa maneira.

O algoritmo DFS tem informações suficientes para classificar algumas arestas à medida que as encontra. A ideia fundamental é que, quando exploramos uma aresta (u, v) pela primeira vez, a cor do vértice v nos diga algo sobre a aresta:

1. BRANCO indica uma aresta de árvore.
2. CINZENTO indica uma aresta de retorno.
3. PRETO indica uma aresta direta ou cruzada.

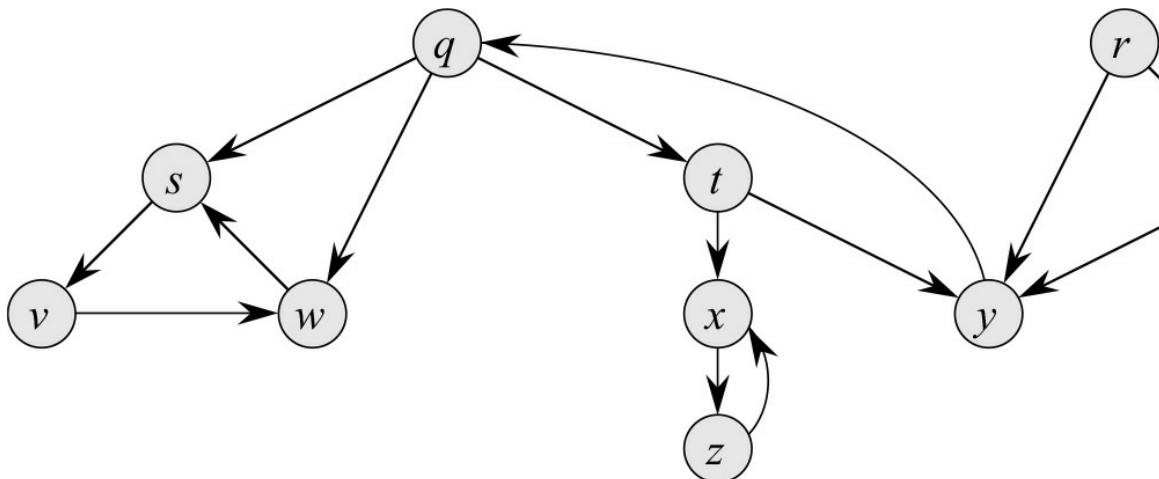
O primeiro caso é imediato pela especificação do algoritmo. Para o segundo caso, observe que os vértices cinzentos sempre formam uma cadeia linear de descendentes que corresponde à pilha de invocações ativas de DFS-VISIT; o número de vértices cinzentos é uma unidade maior que a profundidade na floresta em profundidade do vértice mais recentemente descoberto. A exploração sempre prossegue do vértice cinzento mais profundo; assim, uma aresta que visita outro vértice cinzento alcançou um ancestral. O terceiro caso trata da possibilidade restante; o Exercício 22.3-5 pede que você mostre que tal aresta (u, v) é uma aresta direta se $u.d < v.d$ e uma aresta cruzada se $u.d > v.d$.

Um grafo não dirigido pode acarretar alguma ambiguidade na classificação de arestas, já que (u, v) e (v, u) são na realidade a mesma aresta. Nesse caso, classificamos a aresta como do *primeiro* tipo na lista de classificação aplicável. De modo equivalente (veja o Exercício 22.3-6), classificamos a aresta conforme o que a busca encontrar primeiro: (u, v) ou (v, u) .

Agora, mostramos que arestas diretas e cruzadas nunca ocorrem em uma busca em profundidade de um grafo não dirigido.

Teorema 22.10

Em uma busca em profundidade de um grafo não dirigido G, toda aresta de G é uma aresta de árvore ou é uma aresta de retorno.

Prova Seja (u, v) uma aresta arbitrária de G e suponha, sem perda de generalidade, que $d[u] < d[v]$. Então, a busca deve descobrir e terminar v antes de terminar u (enquanto u é cinzento), já que v está na lista de adjacências de u. Se,

na primeira vez que a busca explorar a aresta (u, v), ela estiver na direção de u para v, então v é não descoberto (branco) até esse momento porque, do contrário, a busca já teria explorado essa aresta na direção de v para u. Assim, (u, v) se torna uma aresta de árvore. Se a busca explorar (u, v) primeiro na direção de v para u, então (u, v) é uma aresta de retorno, já que u ainda é cinzento no momento em que a aresta é explorada pela primeira vez.

Veremos várias aplicações desses teoremas nas seções seguintes.

Exercícios

22.3-1 Faça um diagrama 3 por 3 com rótulos de linhas e colunas BRANCO, CINZENTO e PRETO. Em cada célula (i, j), indique se, em qualquer ponto durante uma busca em profundidade de um grafo dirigido, pode existir uma aresta de um vértice de cor i a um vértice de cor j. Para cada aresta possível, indique os tipos de aresta que ela pode ser. Faça um segundo diagrama para busca em profundidade de um grafo não dirigido.

Figura 22.6 Um grafo dirigido para uso nos Exercícios 22.3-2 e 22.5-2.

22.3-2 Mostre como a busca em profundidade funciona no grafo da Figura 22.6. Suponha que o laço **for** das linhas 5–7 do procedimento DFS considera os vértices em ordem alfabética, e que cada lista de adjacências está em ordem alfabética. Mostre os tempos de descoberta e término para cada vértice e a classificação de cada aresta.

22.3-3 Mostre a estrutura parentizada da busca em profundidade da Figura 22.4.

22.3-4 Mostre que usar um único bit para armazenar a cor de cada vértice é suficiente, demonstrando que o procedimento DFS produziria o mesmo resultado se a linha 8 de DFS-VISIT fosse removida.

2.3-5 Mostre que a aresta (u, v) é

a. uma aresta de árvore ou aresta direta se e somente se $u.d < v.d < v.f < u.f$.

b. uma aresta de retorno se e somente se $v.d < u.d < u.f < v.f$.

c. uma aresta cruzada se e somente se $v.d < v.f < u.d < u.f$.

22.3-6 Mostre que, em um grafo não dirigido, classificar uma aresta (u, v) como uma aresta de árvore ou uma aresta de retorno conforme (u, v) ou (v, u) seja encontrado em primeiro lugar durante a busca em profundidade equivale a classificá-la de acordo com a ordenação dos quatro tipos no esquema de classificação.

22.3-7 Reescreva o procedimento DFS utilizando uma pilha para eliminar recursão.

22.3-8 Dê um contraexemplo para a seguinte hipótese: se um grafo dirigido G contém um caminho de u a v e se $u.d < v.d$ em uma busca em profundidade de G, então v é um descendente de u na floresta em profundidade produzida.

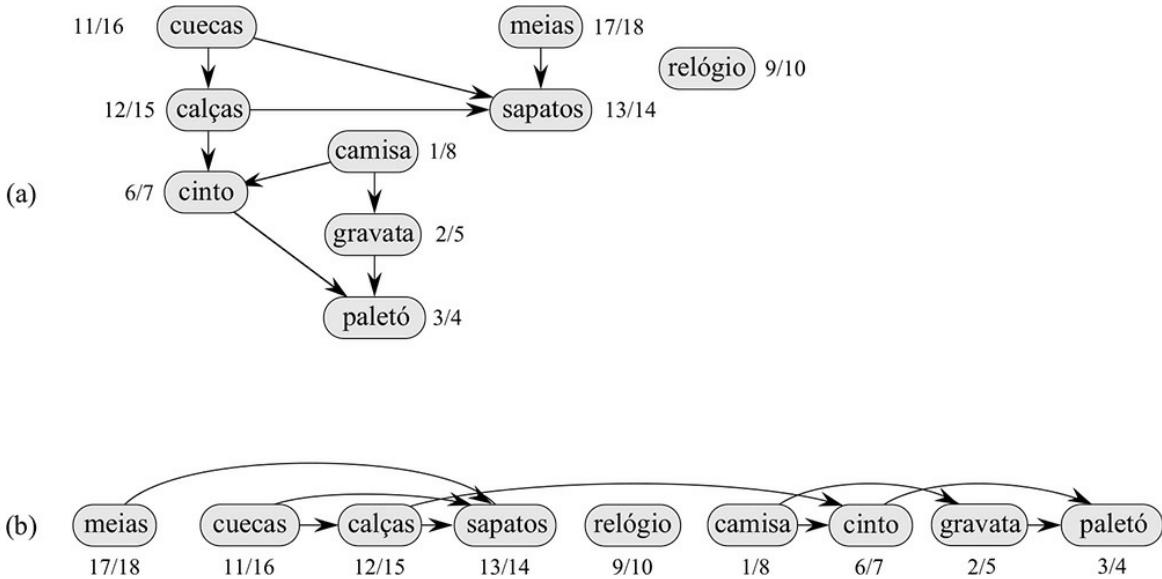
22.3-9 Dê um contraexemplo para a seguinte hipótese: se um grafo dirigido G contém um caminho de u a v, então qualquer busca em profundidade deve resultar em $v.d \leq u.f$.

22.3-10 Modifique o pseudocódigo para busca em profundidade de modo que ele imprima todas as arestas no grafo dirigido G, juntamente com seu tipo. Mostre quais modificações, se houver, você precisa fazer se G for não dirigido.

22.3-11 Explique como um vértice u de um grafo dirigido pode acabar em uma árvore em profundidade que contém apenas u, ainda que u tenha arestas de entrada, bem como de saída em G.

22.3-12 Mostre que podemos usar uma busca em profundidade em um grafo não dirigido G para identificar as componentes conexas de G e que a floresta de busca em profundidade contém tantas árvores quantas são as componentes conexas de G. Mais precisamente, mostre como modificar a busca em profundidade de modo a atribuir a cada vértice v um rótulo inteiro $cc[v]$ entre 1 e k, onde k é o número de componentes conexas de G, tal que $u.cc = v.cc$ se e somente se u e v estiverem na mesma componente conexa.

22.3-13 ★ Um grafo dirigido $G = (V, E)$ é ***singularmente conexo*** se u v implica que G contém no máximo um caminho simples de u a v para todos os vértices $u, v \in V$. Dê um algoritmo eficiente para determinar se um grafo dirigido é ou não isoladamente conexo.

22.4 ORDENAÇÃO TOPOLÓGICA

Esta seção mostra como podemos usar busca em profundidade para executar uma ordenação topológica de um grafo acíclico dirigido, ou “gad”, como, às vezes, é chamado. Uma ***ordenação topológica*** de um gad $G = (V, E)$ é uma ordenação linear de todos os seus vértices, tal que se G contém uma aresta (u, v), então u aparece antes de v na ordenação. (Se o grafo contém um ciclo, nenhuma ordenação topológica é possível.) Podemos ver uma ordenação topológica de um grafo como uma ordenação de seus vértices ao longo de uma linha horizontal de modo tal que todas as arestas dirigidas vão da esquerda para a direita. Assim, a ordenação topológica é diferente do tipo habitual de “ordenação” estudado na Parte II.

Muitas aplicações usam grafos acíclicos dirigidos para indicar precedências entre eventos. A Figura 22.7 mostra um exemplo que surge quando o professor Bumstead se veste pela manhã. O professor deve vestir certas peças de roupa antes de outras (por exemplo, meias antes de sapatos). Outros itens podem ser colocados em qualquer ordem (por exemplo, meias e calças). Uma aresta dirigida (u, v) no gad da Figura 22.7(a) indica que a peça de roupa u deve ser vestida antes da peça v. Portanto, uma ordenação topológica desse gad dá uma ordem para o processo de se vestir. A Figura 22.7(b) mostra o gad topologicamente ordenado como uma ordenação de vértices ao longo de uma linha horizontal tal que todas as arestas dirigidas vão da esquerda para a direita.

O seguinte algoritmo simples ordena topologicamente um gad:

TOPOLOGICAL-SORT(G)

1 chamar DFS(G) para calcular o tempo de término $v.f$ para cada vértice v

2 à medida que cada vértice é terminado, inserir o vértice à frente de uma lista ligada

3 **return** a lista ligada de vértices

Figura 22.7 (**a**) O professor Bumstead ordena topologicamente sua roupa ao se vestir. Cada aresta dirigida (u, v) significa que a peça de roupa u deve ser vestida antes da peça v. Os tempos de descoberta e término de uma busca em profundidade são mostrados ao lado de cada vértice. (**b**) O mesmo grafo mostrado com uma ordenação topológica. Seus vértices estão organizados da esquerda para a direita, em ordem decrescente de tempo de término. Observe que todas as arestas dirigidas vão da esquerda para a direita.

A Figura 22.7(b) mostra como os vértices topologicamente ordenados aparecem na ordem inversa de seus tempos de término.

Podemos executar uma ordenação topológica no tempo $Q(V + E)$, já que a busca em profundidade demora o tempo $Q(V + E)$ e que inserir cada um dos $|V|$ vértices à frente da lista ligada leva o tempo $O(1)$.

Demonstramos a correção desse algoritmo utilizando o seguinte lema fundamental que caracteriza grafos acíclicos dirigidos.

Lema 22.11

Um grafo dirigido G é acíclico se e somente se uma busca em profundidade de G não produz nenhuma aresta de retorno.

Prova ⇒: Suponha que uma busca em profundidade produza uma aresta de retorno (u, v). Então o vértice v é um ancestral do vértice u na floresta em profundidade. Assim, G contém um caminho de v a u, e a aresta de retorno (u, v) completa um ciclo.

⇐: Suponha que G contenha um ciclo c. Mostramos que uma busca em profundidade de G produz uma aresta de retorno. Seja v o primeiro vértice a ser descoberto em c e seja (u, v) a aresta precedente em c. No tempo $v.d$, os vértices de c formam um caminho de vértices brancos de v a u. Pelo teorema do caminho branco, o vértice u se torna um descendente de v na floresta em profundidade. Então, (u, v) é uma aresta de retorno.

Teorema 22.12

TOPOLOGICAL-SORT produz uma ordenação topológica de um grafo acíclico dirigido dado como sua entrada.

Prova Suponha que DFS seja executado em determinado gad $G = (V, E)$ para determinar tempos de término para seus vértices. É suficiente mostrar que, para qualquer par de vértices distintos $u, v \in V$, se G contém uma aresta de u a v, então $v.f < u.f$. Considere qualquer aresta (u, v) explorada por DFS(G). Quando essa aresta é explorada, v não pode ser cinzento, já que nesse caso v seria um ancestral de u e (u, v) seria uma aresta de retorno, o que contradiz o Lema 22.11. Portanto, v deve ser branco ou preto. Se v é branco, ele se torna um descendente de u e, assim, $v.f < u.f$. Se v é preto, ele já terminou, de modo que $v.f$ já foi definido. Como ainda estamos explorando u, ainda temos de atribuir um carimbo de tempo a $u.f$ e, tão logo o façamos, também teremos $v.f < u.f$. Assim, para qualquer aresta (u, v) no gad, temos $v.f < u.f$, o que prova o teorema.

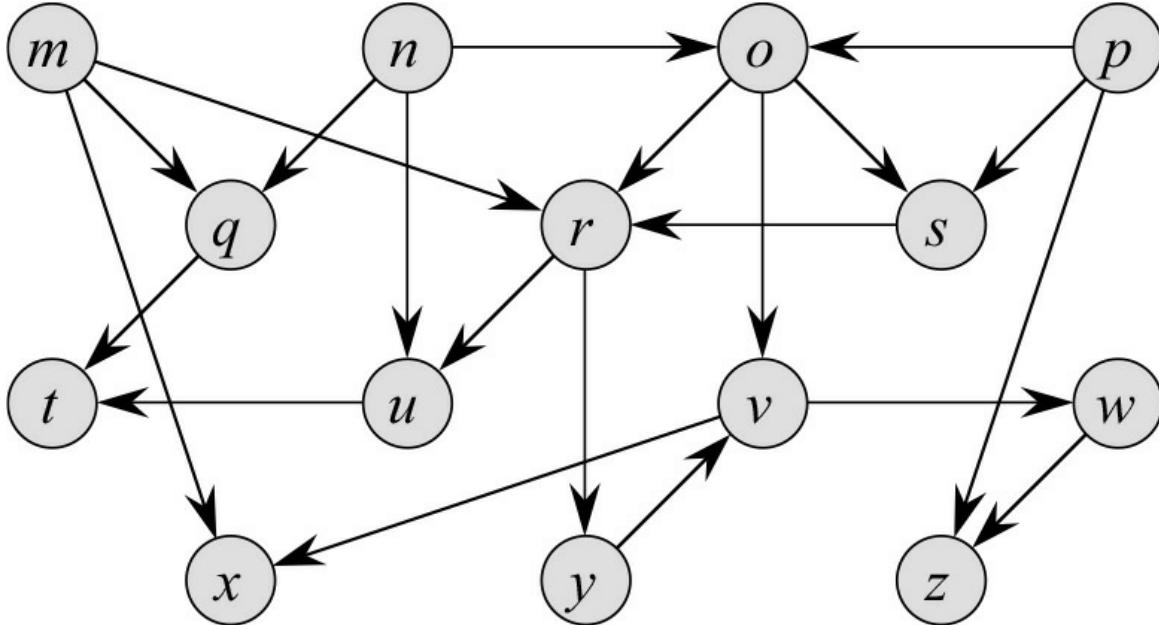



Figura 22.8 Um gad para ordenação topológica.

Exercícios

22.4-1 Mostre a ordenação de vértices produzida por TOPOLOGICAL-SORT quando executado no gad da Figura 22.8, sob a hipótese do Exercício 22.3-2.

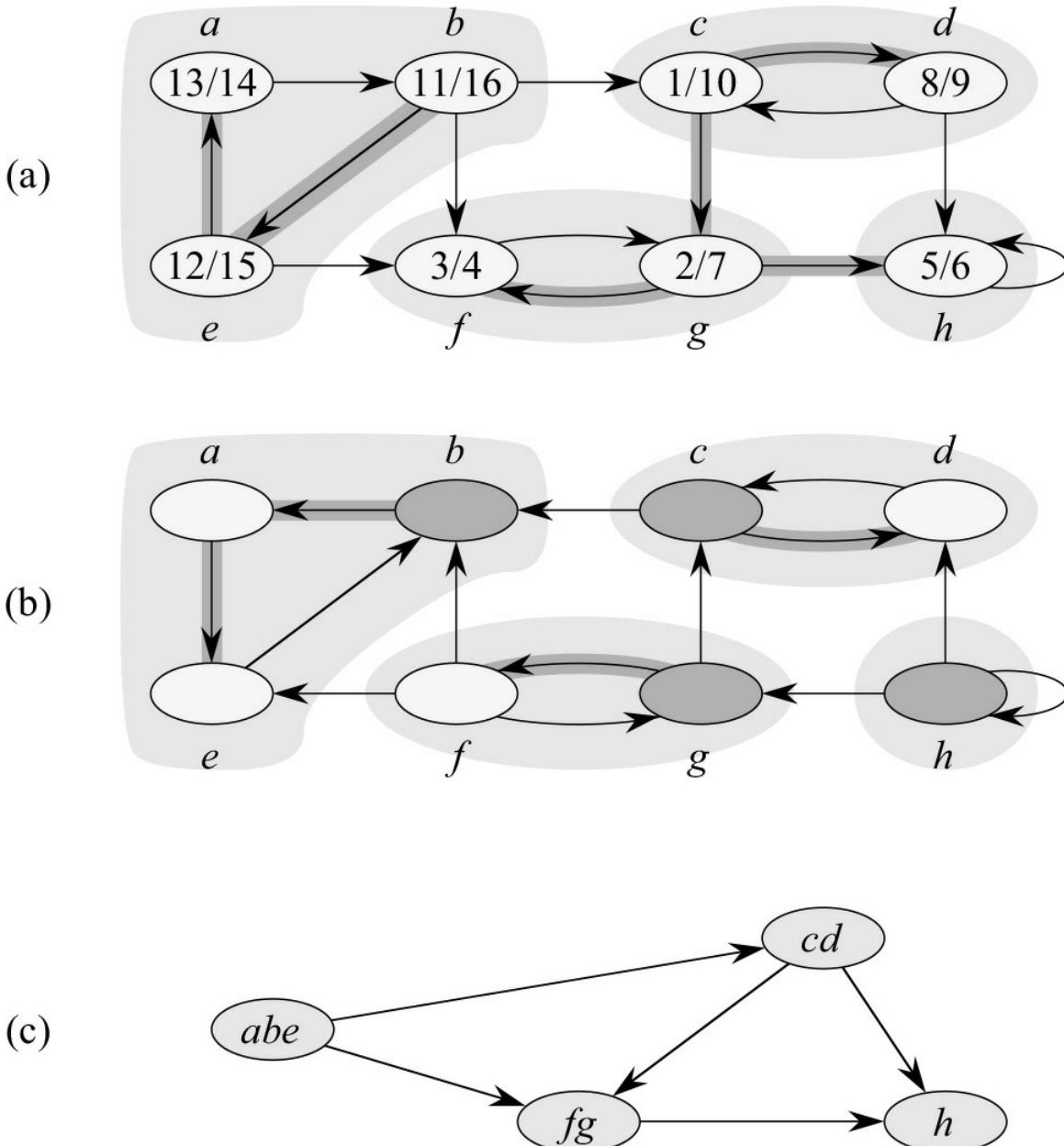
22.4-2 Dê um algoritmo de tempo linear que tome como entrada um grafo acíclico dirigido $G = (V, E)$ e dois vértices s e t, e retorne o número de caminhos simples de s para t em G. Por exemplo, o grafo acíclico dirigido da Figura 22.8 contém exatamente quatro caminhos do vértice p para o vértice v: *pov*, *poryv*, *posryv* e *psryv*. (Seu algoritmo só precisa contar os caminhos, não listá-los.)

22.4-3 Dê um algoritmo que determine se um dado grafo não dirigido $G = (V, E)$ contém um ciclo simples. Esse algoritmo deve ser executado no tempo $O(V)$, independentemente de $|E|$.

22.4-4 Prove ou desprove: se um grafo dirigido G contém ciclos, então TOPOLOGICAL-SORT(G) produz uma ordenação de vértices que minimiza o número de arestas "ruins" que são inconsistentes com a ordenação produzida.

22.4-5 Outro modo de executar ordenação topológica em um grafo acíclico dirigido $G = (V, E)$ é encontrar repetidamente um vértice de grau de entrada 0, imprimi-lo e removê-lo do grafo, bem como todas as suas arestas de saída. Explique como implementar essa ideia, de modo que seja executada no tempo $O(V + E)$. O que acontecerá a esse algoritmo se G tiver ciclos?

22.5 Componentes fortemente conexas

Agora, consideraremos uma aplicação clássica de busca em profundidade: a decomposição de um grafo dirigido em suas componentes fortemente conexas. Esta seção mostra como fazer isso usando duas buscas em profundidade. Muitos algoritmos que funcionam com grafos dirigidos começam por uma decomposição desse tipo. Após a decomposição do grafo em componentes fortemente conexas, tais algoritmos são executados separadamente em cada uma delas e combinados em soluções de acordo com a estrutura das conexões entre componentes.

Lembre-se de que vimos no Apêndice B que uma componente fortemente conexa de um grafo dirigido $G = (V, E)$ é um conjunto máximo de vértices $C \subseteq V$ tal que, para todo par de vértices u e v em C, temos u v e v u; isto é, u pode ser alcançado a partir do vértice v e vice-versa. A Figura 22.9 mostra um exemplo.

Nosso algoritmo para encontrar componentes fortemente conexas de um grafo $G = (V, E)$ usa o transposto de G, que é definida no Exercício 22.1-3 como o grafo $G_T = (V, E_T)$, onde $E_T = \{(u, v) : (v, u) \in E\}$. Isto é, E_T consiste nas arestas de G com suas direções invertidas. Dada uma representação por lista de adjacências de G, o tempo para criar G_T é $O(V + E)$. É interessante observar que G e G_T têm exatamente as mesmas componentes fortemente conexas: u e v, podem ser alcançados um a partir do outro em G se e somente se puderem ser alcançados um a partir do outro em G_T. A Figura 22.9(b) mostra o transposto do grafo na Figura 22.9(a), com as componentes fortemente conexas sombreadas.

O algoritmo de tempo linear (isto é, de tempo Q($V + E$)) apresentado a seguir calcula as componentes fortemente conexas de um grafo dirigido $G = (V, E)$ usando duas buscas em profundidade, uma em G e uma em G_T.

```
STRONGLY-CONNECTED-COMPONENTS(G).
1    chamar DFS(G) para calcular tempos de término u.f para cada vértice u
2    calcular G^T
3    chamar DFS(G^T) mas, no laço principal de DFS, considerar os vértices em ordem
         decrescente de u.f (como calculado na linha 1)
4    dar saída aos vértices de cada árvore na floresta em profundidade formada na linha 3
         como uma componente fortemente conexa separada
```

Figura 22.9 (**a**) Um grafo dirigido G. Cada região sombreada é uma componente fortemente conexa de G. Cada vértice é identificado com seus tempos de descoberta e de término em uma busca em profundidade, e arestas de árvore são sombreadas. (**b**) O grafo G_T, a transposta de G, na qual é mostrada a árvore em profundidade calculada na linha 3 de STRONGLY-CONNECTED-COMPONENTS e as arestas de árvore são sombreadas. Cada componente fortemente conexa corresponde a uma árvore de busca em profundidade. Os vértices b, c, g e h, que são sombreados em tom mais escuro, são as raízes das árvores de busca em profundidade produzidas pela busca em profundidade de G_T. (**c**) O grafo acíclico de componentes G_{SCC} obtido pela contração de cada componente fortemente conexa de G, de modo que apenas um único vértice permaneça em cada componente.

A ideia por trás desse algoritmo vem de uma propriedade fundamental do ***grafo de compo- nentes*** $G_{SCC} = (V_{SCC}, E_{SCC})$, que definimos a seguir. Suponha que G tenha componentes fortemente conexas $C_1, C_2, ..., C_k$. O conjunto de vértices V_{SCC} é $\{v_1, v_2, ..., v_k\}$ e contém um vértice v_i para cada componente fortemente conexa C_i de G. Há uma aresta $(v_i, v_j) \in E_{SCC}$ se G contém uma aresta dirigida (x, y) para algum $x \in C_i$ e algum $y \in C_j$. Visto de outro modo, contraindo todas as arestas cujos vértices incidentes estão dentro da mesma componente fortemente conexa de G, o grafo resultante é G_{SCC}. A Figura 22.9(c) mostra o grafo de componentes do grafo na Figura 22.9(a).

A propriedade fundamental é que o grafo de componentes é um gad, o que implica o lema a seguir.

Lema 22.13

Sejam C e C' componentes fortemente conexas distintas em um grafo dirigido $G = (V, E)$, seja $u, v \in C$, seja $u', v' \in C'$ e suponha que G contenha um caminho $u \;\; u'$. Então, G não pode conter também um caminho $v' \;\; v$.

Prova Se G contém um caminho $v' \;\; v$, então contém caminhos $u \;\; u' \;\; v'$ e $v' \;\; v \;\; u$ em G. Assim, u e v' podem ser visitados um a partir do outro, o que contradiz a hipótese de que C e C' são componentes fortemente conexas distintas.

Veremos que, considerando vértices na segunda busca em profundidade em ordem decrescente dos tempos de término que foram calculados na primeira busca em profundidade, estamos, em essência, visitando os vértices do grafo de componentes (cada um dos quais corresponde a uma componente fortemente conexa de G) em sequência ordenada topologicamente.

Como o procedimento STRONGLY-CONNECTED-COMPONENTS executa duas buscas em profundidade, há potencial para ambiguidade quando discutimos $u.d$ ou $u.f$. Nesta seção, esses valores sempre se referem aos tempos de descoberta e término calculados pela *primeira* chamada de DFS, na linha 1.

Estendemos a notação de tempos de descoberta e término a conjuntos de vértices. Se $U \subseteq V$, então definimos $d(U) = \min_{u \in U} \{u.d\}$ e $f(U) = \max_{u \in U} \{u.f\}$. Isto é, $d(U)$ e $f(U)$ são o tempo de descoberta mais antigo e o tempo de término mais recente, respectivamente, de qualquer vértice em U.

O lema a seguir e seu corolário dão uma propriedade fundamental que relaciona componentes fortemente conexas a tempos de término na primeira busca em profundidade.

Lema 22.14

Sejam C e C' componentes fortemente conexas distintas no grafo dirigido $G = (V, E)$. Suponha que haja uma aresta $(u, v) \in E$, onde $u \in C$ e $v \in C'$. Então, $f(C) > f(C')$.

Prova Consideramos dois casos, dependendo de qual componente fortemente conexa, C ou C', tinha o primeiro vértice descoberto durante a busca em profundidade.

Se $d(C) < d(C')$, seja x o primeiro vértice descoberto em C. No tempo $x.d$, todos os vértices

em C e C' são brancos. Nesse momento, G contém um caminho de x a cada vértice em C ao longo do qual há apenas vértices brancos. Como $(u, v) \in E$, para qualquer vértice $w \in C'$, também há em G um caminho de x a w no tempo $x.d$ que contém somente vértices brancos: $x \;\; u \to v \;\; w$. Pelo teorema do caminho branco, todos os vértices em C e C' se tornam descendentes de x na árvore em profundidade. Pelo Corolário 22.8, x tem o tempo de término mais recente que qualquer de seus descendentes, portanto, $x.f = f(C) > f(C')$.

Se, em vez disso, tivermos $d(C) > d(C')$, seja y o primeiro vértice descoberto em C'. No tempo $y.d$, todos os vértices em C' são brancos e G contém um caminho de y a cada vértice em C' formado somente por vértices brancos. Pelo teorema do caminho branco, todos os vértices em C' se tornam descendentes de y na árvore em profundidade, e pelo Corolário 22.8, $y.f = f(C')$. No tempo $y.d$, todos os vértices em C são brancos. Como existe uma aresta (u, v) de C a C', o Lema 22.13 implica que não pode existir um caminho de C' a C. Consequentemente, nenhum vértice em C pode ser visitado por y. Portanto, no tempo $y.f$, todos os vértices em C ainda são brancos. Assim, para qualquer vértice $w \in C$, temos $w.f > y.f$, o que implica que $f(C) > f(C')$.

O corolário a seguir nos diz que cada aresta em G_T entre diferentes componentes fortemente conexas vai de uma componente com um tempo de término anterior (na primeira busca em profundidade) a uma componente com um tempo de término posterior.

Corolário 22.15

Sejam C e C' componentes fortemente conexas distintas no grafo dirigido $G = (V, E)$. Suponha que haja uma aresta $(u, v) \in E_T$, onde $u \in C$ e $v \in C'$. Então, $f(C) < f(C')$.

Prova Como $(u, v) \in E_T$, temos $(v, u) \in E$. Visto que as componentes fortemente conexas de G e G_T são as mesmas, o Lema 22.14 implica que $f(C) < f(C')$.

O Corolário 22.15 nos dá a chave para entender por que o algoritmo de componentes fortemente conexas funciona. Vamos examinar o que acontece quando executamos a segunda busca em profundidade, que está em G_T. Começamos com a componente fortemente conexa C cujo tempo de término $f(C)$ é máximo. A busca começa em algum vértice $x \in C$ e visita todos os vértices em C. Pelo Corolário 22.15, G_T não contém nenhuma aresta de C a qualquer outra componente fortemente conexa e, por isso, a busca iniciada em x não visitará vértices em qualquer outra componente. Assim, a árvore com raiz em x contém exatamente os vértices de C. Agora que as visitas a todos os vértices em C foram concluídas, a busca na linha 3 seleciona como raiz um vértice de alguma outra componente fortemente conexa C' cujo tempo de término $f(C')$ é máximo em relação a todas as outras componentes, exceto C. Mais uma vez, a busca visitará todos os vértices em C' mas, pelo Corolário 22.15, as únicas arestas em G_T que vão de C' a qualquer outra componente devem ir até C, que já visitamos. Em geral, quando a busca em profundidade de G_T na linha 3 visita qualquer componente fortemente conexa, quaisquer arestas que saem dessa componente devem ir até componentes que a busca já visitou. Então, cada árvore de busca em profundidade será exatamente uma componente fortemente conexa. O teorema a seguir formaliza esse argumento.

Teorema 22.16

O procedimento STRONGLY-CONNECTED-COMPONENTS calcula corretamente as componentes fortemente conexas do grafo dirigido dado como sua entrada.

Prova Mostramos por indução em relação ao número de árvores de busca encontradas na busca em profundidade de G_T na linha 3 que os vértices de cada árvore formam uma componente fortemente conexa. A hipótese de indução é que as primeiras k árvores produzidas na linha 3 são componentes fortemente conexas. A base para a indução, quando $k = 0$, é trivial.

No passo de indução, supomos que cada uma das k primeiras árvores em profundidade produzidas na linha 3 é uma componente fortemente conexa, e consideramos a $(k + 1)$-ésima árvore produzida. Seja o vértice u a raiz dessa árvore, e suponhamos que u esteja na componente fortemente conexa C. Como resultado do modo como escolhemos raízes na busca em profundidade na linha 3, $u.f = f(C) > f(C')$ para qualquer componente fortemente conexa C' exceto C que ainda tenha de ser visitada. Pela hipótese de indução, no momento em que a busca visita u, todos os outros vértices de C são brancos. Então, pelo teorema do caminho branco, todos os outros vértices de C são descendentes de u nessa árvore em profundidade. Além disso, pela hipótese de indução e pelo Corolário 22.15, quaisquer arestas em G_T que saem de C devem ir até componentes fortemente conexas que já foram visitadas. Assim, nenhum vértice em uma componente fortemente conexa exceto C será um descendente de u durante a busca em profundidade de G_T. Portanto, os vértices da árvore de busca em profundidade em G_T enraizada em u formam exatamente uma componente fortemente conexa, o que conclui o passo de indução e a prova.

Apresentamos agora um outro modo de ver como funciona a segunda busca em profundidade. Considere o grafo de componentes $(G_T)^{SCC}$ de G_T. Se mapearmos cada componente fortemente conexa visitada na segunda busca em profundidade até um vértice de $(G_T)^{SCC}$, a segunda busca em profundidade visita os vértices de $(G_T)^{SCC}$ na ordem inversa de uma ordem topológica. Se invertermos as arestas de $(G_T)^{SCC}$, obteremos o grafo $((G_T)^{SCC})^T$. Como $((G_T)^{SCC})^T = (G_T)^{SCC}$ (veja o Exercício 22.5-4), a segunda busca em profundidade visita os vértices de G_{SCC} em ordem topológica.

Exercícios

22.5-1 Como o número de componentes fortemente conexas de um grafo pode mudar se uma nova aresta for adicionada?

22.5-2 Mostre como o procedimento STRONGLY-CONNECTED-COMPONENTS funciona no grafo da Figura 22.6. Especificamente, mostre os tempos de término calculados na linha 1 e a floresta produzida na linha 3. Suponha que o laço das linhas 5–7 de DFS toma os vértices em ordem alfabética e que as listas de adjacências estão em ordem alfabética.

22.5-3 O professor Bacon afirma que o algoritmo para componentes fortemente conexas seria mais simples se usasse o grafo original (em lugar da transposta) na segunda busca em profundidade e varresse os vértices na ordem *crescente* de tempos de término. Esse algoritmo mais simples sempre produzirá resultados corretos?

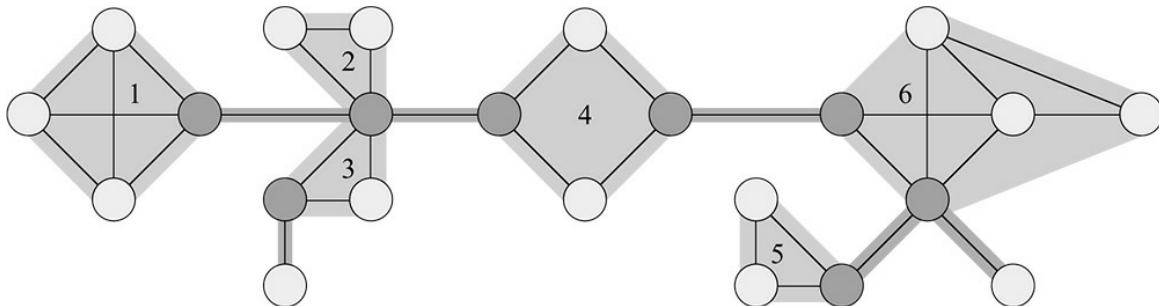
22.5-4 Prove que, para qualquer grafo dirigido G, temos $((G_T)^{SCC})^T = (G_T)^{SCC}$. Isto é, o transposto do grafo de componentes de G_T é igual ao grafo de componentes de G.

22.5-5 Dê um algoritmo de tempo $O(V + E)$ para calcular o grafo de componentes de um grafo dirigido $G = (V, E)$. Certifique-se de que haja no máximo uma aresta entre dois vértices no grafo de componentes que o seu algoritmo produz.

22.5-6 Dado um grafo dirigido $G = (V, E)$, explique como criar um outro grafo $G' = (V, E')$ tal que (a) G' tenha as mesmas componentes fortemente conexas que G, (b) G' tenha o mesmo grafo de componentes que G, e (c) E' seja o menor possível. Descreva um algoritmo rápido para calcular G'.

22.5-7 Um grafo dirigido $G = (V, E)$ é ***semiconexo*** se, para todos os pares de vértices $u, v \in V$, temos u v ou v u. Dê um algoritmo eficiente para determinar se G é ou não semiconexo. Prove que o algoritmo é correto e analise seu tempo de execução.

Problemas

22-1 ***Classificação de arestas por busca em largura***

Uma floresta de busca em profundidade classifica as arestas de um grafo em arestas de árvore, de retorno, diretas e cruzadas. Uma árvore de busca em largura também pode ser usada para classificar as arestas que podem ser alcançadas a partir da fonte da busca nas mesmas quatro categorias.

a. Prove que, em uma busca em largura de um grafo não dirigido, as seguintes propriedades são válidas:

1. Não há nenhuma aresta de retorno e nenhuma aresta direta.
2. Para cada aresta de árvore (u, v), temos $v.d = u.d + 1$.
3. Para cada aresta cruzada (u, v), temos $v.d = u.d$ ou $v.d = u.d + 1$.

b. Prove que, em uma busca em largura de um grafo dirigido, as seguintes propriedades são válidas:

1. Não há arestas diretas.
2. Para cada aresta de árvore (u, v), temos $v.d = u.d + 1$.
3. Para cada aresta cruzada (u, v), temos $v.d \leq u.d + 1$.
4. Para cada aresta de retorno (u, v), temos $0 \leq v.d < u.d$.

22-2 Pontos de articulação, pontes e componentes biconectadas

Seja $G = (V, E)$ um grafo conexo não dirigido. Um ***ponto de articulação*** de G é um vértice cuja remoção desconecta G. Uma ***ponte*** de G é uma aresta cuja remoção desconecta G. Uma ***componente biconexa*** de G é um conjunto máximo de arestas tal que quaisquer duas arestas no conjunto encontram-se em um ciclo simples comum. A Figura 22.10 ilustra essas definições. Podemos determinar pontos de articulação, pontes e componentes biconexas utilizando busca em profundidade. Seja $G_p = (V, E_p)$ uma árvore de busca em profundidade de G.

a. Prove que a raiz de G_p é um ponto de articulação de G se e somente se ele tem no mínimo dois filhos em G_p.

Figura 22.10 Os pontos de articulação, pontes e componentes biconexas de um grafo conexo não dirigido para uso no Problema 22-2. Os pontos de articulação são os vértices sombreados em tom mais escuro, as pontes são as arestas sombreadas em tom mais escuro e as componentes biconexas são as arestas nas regiões sombreadas, nas quais aparece a numeração *bcc*.

b. Seja v um vértice não de raiz de G_p. Prove que v é um ponto de articulação de G se e somente se v tem um filho s tal que não há nenhuma aresta de retorno de s ou de qualquer descendente de s até um ancestral próprio de v.

c. Seja

$$v.inferior = \min \begin{cases} v.d, \\ w.d : (u, w) \text{ é uma aresta de retorno para algum descendente } u \text{ de } v. \end{cases}$$

Mostre como calcular $v.inferior$ para todos os vértices $v \in V$ no tempo $O(E)$.

d. Mostre como calcular todos os pontos de articulação no tempo $O(E)$.

e. Prove que uma aresta de G é uma ponte se e somente se ela não estiver em nenhum ciclo simples de G.

f. Mostre como calcular todas as pontes de G no tempo $O(E)$.

g. Prove que as componentes biconexas de G particionam as arestas não pontes de G.

h. Dê um algoritmo de tempo $O(E)$ para rotular cada aresta e de G com um inteiro positivo $e.bcc$ tal que $e.bcc = e'.bcc$ se e somente se e e e' estão na mesma componente biconexa.

22-3 Percurso de Euler

Um ***percurso de Euler*** de um grafo fortemente conexo dirigido $G = (V, E)$ é um ciclo que percorre cada aresta de G exatamente uma vez, embora possa visitar um vértice mais de uma vez.

a. Mostre que G tem um percurso de Euler se e somente se grau de entrada(v) = grau de saída(v) para cada vértice $v \in V$.

b. Descreva um algoritmo de tempo $O(E)$ para encontrar um percurso de Euler de G se houver algum. (*Sugestão*: Intercale ciclos disjuntos de arestas.)

22-4 Acessibilidade

Seja $G = (V, E)$ um grafo dirigido no qual cada vértice $u \in V$ é rotulado com um inteiro único $L(u)$ do conjunto $\{1, 2, ..., |V|\}$. Para cada vértice $u \in V$, seja $R(u) = \{v \in V : u \quad v\}$ o conjunto de vértices que podem ser alcançados a partir de u. Defina min(u) como o vértice em $R(u)$ cujo rótulo é mínimo, isto é, min(u) é o vértice v tal que $L(v) = \min\{L(w) : w \in R(u)\}$. Dê um algoritmo de tempo $O(V + E)$ que calcule min(u) para todos os vértices $u \in V$.

NOTAS DO CAPÍTULO

Even [103] e Tarjan [330] são excelentes referências para algoritmos em grafos.

A busca em largura foi descoberta por Moore [260] no contexto de caminhos de localização em labirintos. Lee [226] descobriu independentemente o mesmo algoritmo no contexto de roteamento de fios em placas de circuitos.

Hopcroft e Tarjan [178] defenderam o uso da representação por listas de adjacências em vez da representação por matriz de adjacências, no caso de grafos esparsos, e foram os primeiros a reconhecer a importância algorítmica da busca em profundidade. A busca em profundidade tem sido amplamente utilizada desde o final da década de 1950, especialmente em programas de inteligência artificial.

Tarjan [327] apresentou um algoritmo de tempo linear para encontrar componentes fortemente conexas. O algoritmo para componentes fortemente conexas na Seção 22.5 foi adaptado de Aho, Hopcroft e Ullman [6], que o creditam a S. R. Kosaraju (não publicado) e M. Sharir [314]. Gabow [119] também desenvolveu um algoritmo para componentes fortemente conexas baseado na contração de ciclos, e utiliza duas pilhas para executá-lo em tempo linear. Knuth [209] foi o primeiro a apresentar um algoritmo de tempo linear para ordenação topológica.

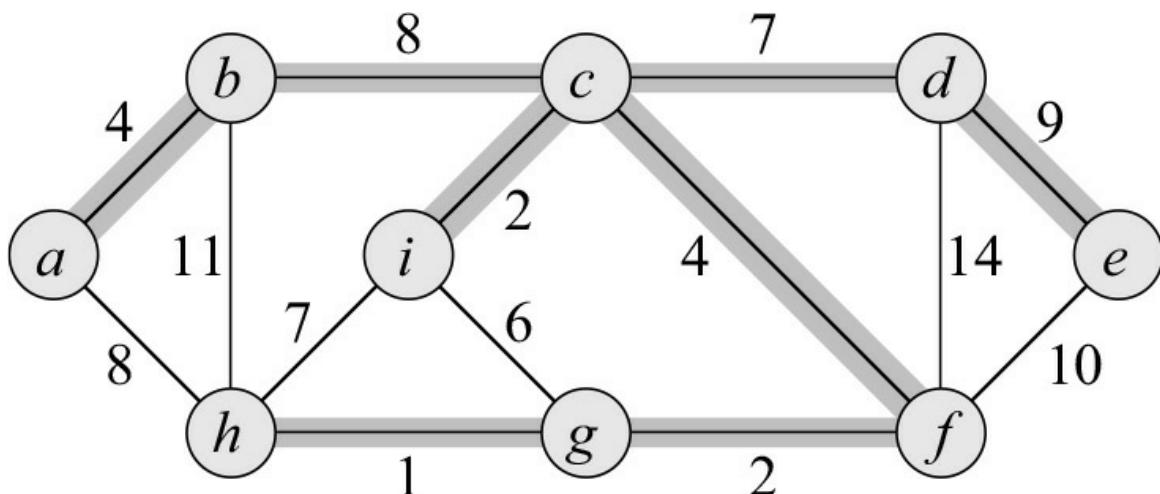
---

[1]Distiguimos entre vértices cinzentos e pretos porque isso nos ajuda a entender como a busca em largura funciona. Na verdade, como o Exercício 22.2-3 mostra, obteríamos o mesmo resultado mesmo que não distinguíssemos entre vértices cinzentos e pretos.

[2]Nos Capítulos 24 e 25, generalizaremos nosso estudo de caminhos mínimos para grafos ponderados, nos quais cada aresta tem um valor de peso real e o peso de um caminho é a soma dos pesos de suas arestas constituintes. Os grafos considerados neste capítulo são grafos não ponderados ou, o que é equivalente, todas as arestas têm peso unitário.

3 Pode parecer arbitrário que a busca em largura se limite apenas a uma fonte, enquanto a busca em profundidade pode executar busca partindo de várias fontes. Embora, em termos conceituais, a busca em largura poderia se originar em várias fontes e a busca em profundidade poderia ser limitada a uma fonte, nossa abordagem reflete o modo como os resultados dessas buscas são normalmente usados. Em geral, a busca em largura serve para encontrar distâncias de caminhos mínimos (e o subgrafo predecessor associado) que partem de determinada fonte. A busca em profundidade muitas vezes é uma sub-rotina em um outro algoritmo, como veremos mais adiante neste capítulo.

23 ÁRVORES GERADORAS MÍNIMAS

Em projeto de circuitos eletrônicos, muitas vezes, é necessário que os pinos de vários componentes se tornem eletricamente equivalentes, o que é conseguido ligando-os uns aos outros. Para interconectar um conjunto de n pinos, podemos usar um arranjo de $n - 1$ fios, cada qual conectando dois pinos. De todos os arranjos possíveis, aquele que utiliza a mínima quantidade de fio é normalmente o mais desejável.

Podemos modelar esse problema de fiação com um grafo conexo não dirigido $G = (V, E)$, onde V é o conjunto de pinos, E é o conjunto de interconexões possíveis entre pares de pinos e, para cada aresta $(u, v) \in E$, temos um peso $w(u, v)$ que especifica o custo (a quantidade necessária de fio) para conectar u e v. Então, desejamos encontrar um subconjunto acíclico $T \subseteq E$ que conecte todos os vértices e cujo peso total

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

é minimizado. Visto que T é acíclico e conecta todos os vértices, deve formar uma árvore, que denominaremos ***árvore geradora***, já que "gera" o grafo G. O problema de determinar a árvore T é denominado ***problema da árvore geradora mínima***[1]. A Figura 23.1 mostra um exemplo de grafo conexo e uma árvore geradora mínima.

Neste capítulo, examinaremos dois algoritmos para resolver o problema da árvore geradora mínima: o algoritmo de Kruskal e o algoritmo de Prim. É fácil fazer com que cada um deles seja executado no tempo $O(E \lg V)$ utilizando heaps binários comuns. Se usarmos heaps de Fibonacci, o algoritmo de Prim é executado no tempo $O(E + V \lg V)$, o que representa uma melhoria em relação à implementação com heaps binários se $|V|$ é muito menor que $|E|$.

Figura 23.1 Uma árvore geradora mínima para um grafo conexo. Os pesos nas arestas são mostrados, e as arestas em uma árvore geradora mínima estão sombreadas. O peso total da árvore mostrada é 37. Essa árvore geradora mínima não é única: se removermos a aresta (b, c) e a substituirmos pela aresta (a, h) produziremos outra árvore geradora com peso 37.

Os dois algoritmos são algoritmos gulosos, como descreve o Capítulo 16. Cada etapa de um algoritmo guloso deve fazer uma entre várias opções possíveis. A estratégia gulosa faz a escolha que é a melhor no momento. Em geral, tal estratégia não garante que sempre encontrará soluções globalmente ótimas para problemas. Porém, no caso do problema da árvore geradora mínima, podemos provar que certas estratégias gulosas realmente produzem uma árvore geradora com peso mínimo. Embora este capítulo possa ser lido independentemente do Capítulo 16, os métodos gulosos apresentados aqui são uma aplicação clássica das noções teóricas introduzidas naquele capítulo.

A Seção 23.1 introduz um método "genérico" de árvore geradora mínima que produz uma árvore geradora adicionando uma aresta por vez. A Seção 23.2 dá dois algoritmos que implementam o método genérico. O primeiro algoritmo, desenvolvido por Kruskal, é semelhante ao algoritmo de componentes conexas da Seção 21.1. O segundo, desenvolvido por Prim, é semelhante ao algoritmo de caminhos mínimos de Dijkstra (Seção 24.3).

Como uma árvore é um tipo de grafo, se quisermos ser precisos temos de definir uma árvore em termos não apenas de suas arestas, mas também de seus vértices. Embora este capítulo focalize árvores em termos de suas arestas, continuaremos entendendo que os vértices de uma árvore T são aqueles nos quais incide alguma aresta de T.

23.1 DESENVOLVENDO UMA ÁRVORE GERADORA MÍNIMA

Suponha que temos um grafo conexo não dirigido $G = (V, E)$ com uma função peso $w : E \rightarrow$ e desejamos encontrar uma árvore geradora mínima para G. Os dois algoritmos que consideramos neste capítulo utilizam uma abordagem gulosa para o problema, embora os modos como aplicam essa abordagem sejam diferentes.

Essa estratégia gulosa é representada pelo método genérico apresentado a seguir, que desenvolve a árvore geradora mínima uma aresta por vez. O método genérico administra um conjunto de arestas A, mantendo o seguinte invariante de laço:

Antes de cada iteração, A é um subconjunto de alguma árvore geradora mínima.

Em cada etapa, determinamos uma aresta (u, v) que pode ser adicionada a A sem violar esse invariante, no sentido de que $A \cup \{(u, v)\}$ também é um subconjunto de uma árvore geradora mínima. Denominamos tal aresta ***aresta segura*** para A, já que ela pode ser adicionada com segurança a A e, ao mesmo tempo, manter o invariante.

```
GENERIC-MST(G, w)
1    A = ∅
2    while A não formar uma árvore geradora
3         encontre uma aresta (u, v) que seja segura para A
4         A = A ∪ {(u, v)}
5    return A
```

Usamos o invariante de laço da seguinte maneira:

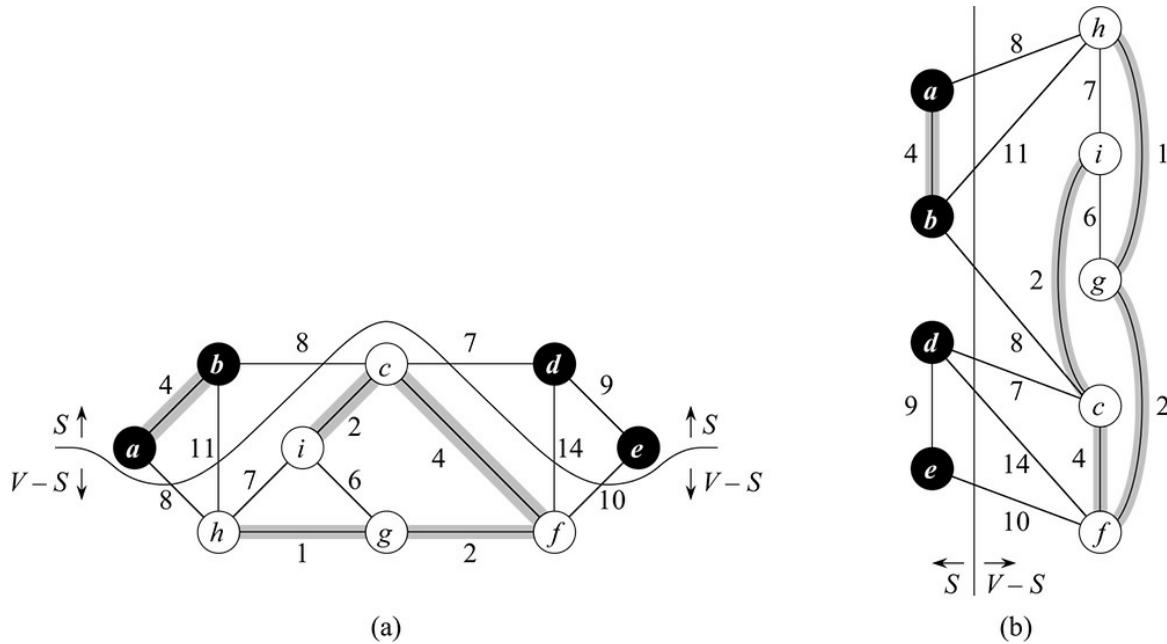
Inicialização: Depois da linha 1, o conjunto A satisfaz trivialmente o invariante de laço.

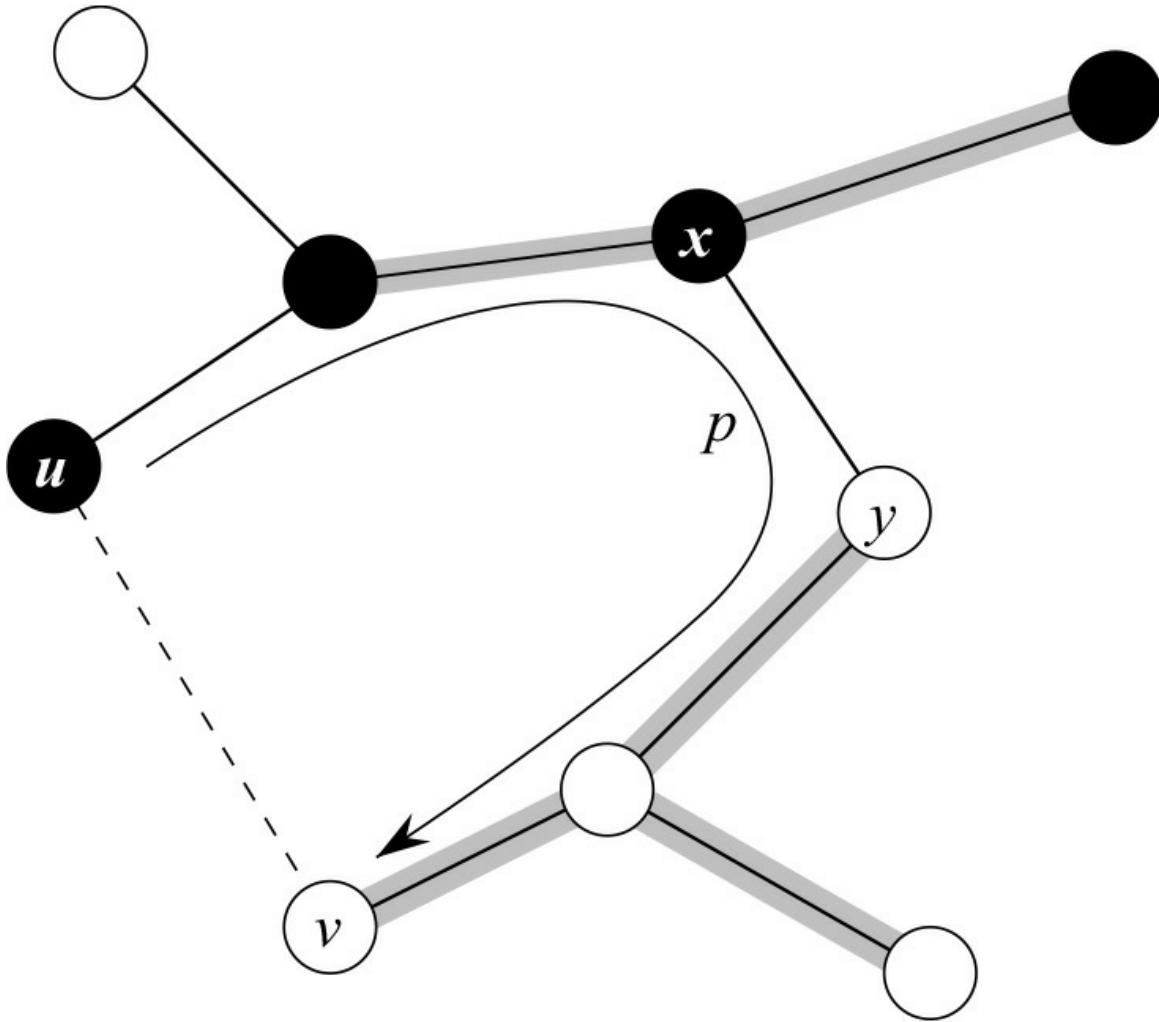
Manutenção: O laço nas linhas 2-4 mantém o invariante, adicionando apenas arestas seguras.

Término: A está contido em uma árvore geradora mínima e uma árvore geradora, portanto, o conjunto A devolvido na linha 5 deve ser uma árvore geradora mínima.

É claro que a parte complicada é encontrar uma aresta segura na linha 3. Deve existir uma, já que, quando a linha 3 é executada, o invariante estabelece que existe uma árvore geradora T tal que $A \subseteq T$. Dentro do corpo do laço **while**, A deve ser um subconjunto próprio de T, e portanto deve haver uma aresta $(u, v) \in T$ tal que $(u, v) \notin A$ e (u, v) é segura para A.

No restante desta seção, daremos uma regra (Teorema 23.1) para reconhecer arestas seguras. A próxima seção descreve dois algoritmos que usam essa regra para encontrar arestas seguras eficientemente.

Primeiro, precisamos de algumas definições. Um ***corte*** $(S, V - S)$ de um grafo não dirigido $G = (V, E)$ é uma partição de V. A Figura 23.2 ilustra essa noção. Dizemos que uma aresta $(u, v) \in E$ ***cruza*** o corte $(S, V - S)$ se um de seus pontos extremos está em S e o outro está em $V - S$. Dizemos que um corte ***respeita*** um conjunto A de arestas se nenhuma aresta em A cruza o corte. Uma aresta é uma ***aresta leve*** que cruza um corte se seu peso é o mínimo de qualquer aresta que cruza o corte. Observe que pode haver mais de uma aresta leve que cruza um corte no caso de empates. De modo mais geral, dizemos que uma aresta é uma ***aresta leve*** que satisfaz uma dada propriedade se seu peso é o mínimo de qualquer aresta que satisfaz a propriedade.

Nossa regra para reconhecer arestas seguras é dada pelo seguinte teorema.

Teorema 23.1

Seja $G = (V, E)$ um grafo conexo não dirigido com uma função peso de valores reais w definida em E. Seja A um subconjunto de E que está incluído em alguma árvore geradora mínima para G, seja $(S, V - S)$ qualquer corte de G que respeita A e seja (u, v) uma aresta leve que cruza $(S, V - S)$. Então, a aresta (u, v) é segura para A.

Prova Seja T uma árvore geradora mínima que inclui A, e suponha que T não contenha a aresta leve (u, v) já que, se contiver, terminamos. Construiremos uma outra árvore geradora mínima T' que inclui $A \cup \{(u, v)\}$ usando uma técnica de recortar e colar, mostrando assim que (u, v) é uma aresta segura para A.
A aresta (u, v) forma um ciclo com as arestas no caminho simples p de u a v em T, como ilustra a Figura 23.3. Visto que u e v estão em lados opostos do corte $(S, V - S)$, no mínimo uma aresta em T se encontra no caminho simples p e também cruza o corte. Seja (x, y) qualquer dessas arestas. A aresta (x, y) não está em A porque o corte respeita A. Como (x, y) está no único caminho simples de u a v em T, remover (x, y) separa T em duas componentes. A adição de (u, v) reconecta as duas componentes e forma uma nova árvore geradora $T' = T - (x, y) \cup \{(u, v)\}$.

Figura 23.2 Duas maneiras de visualizar um corte $(S, V - S)$ do grafo da Figura 23.1. (**a**) Vértices pretos estão no conjunto S e vértices brancos estão em $V - S$. As arestas que cruzam o corte são as que conectam vértices brancos com vértices pretos. A aresta (d, c) é a única aresta leve que cruza o corte. Um subconjunto A das arestas está sombreado; observe que o corte $(S, V - S)$ respeita A, já que nenhuma aresta de A cruza o corte. (**b**) O mesmo grafo com os vértices no conjunto S à esquerda e os vértices no conjunto $V - S$ à direita. Uma aresta cruza o corte se ela conecta o vértice à esquerda com um vértice à direita.

Figura 23.3 A prova do Teorema 23.1. Os vértices pretos estão em S e os vértices brancos estão em $V - S$. As arestas na árvore geradora mínima T são mostradas, mas as arestas no grafo G não são. As arestas em A são sombreadas, e (u, v) é uma aresta leve que cruza o corte $(S, V - S)$. A aresta (x, y) é uma aresta no caminho simples único p de u a v em T. Para formar uma árvore geradora mínima T' que contém (u, v), remova a aresta (x, y) de T e adicione a aresta (u, v).

Em seguida, mostramos que T' é uma árvore geradora mínima. Visto que (u, v) é uma aresta leve que cruza $(S, V - S)$ e (x, y) também cruza esse corte, $w(u, v) \leq w(x, y)$. Então,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) . \end{aligned}$$

Porém, T é uma árvore geradora mínima, de modo que $w(T) \leq w(T')$; assim, T' também deve ser uma árvore geradora mínima.

Resta mostrar que (u, v) é realmente uma aresta segura para A. Temos $A \subseteq T'$, já que $A \subseteq T$ e $(x, y) \notin A$; assim, $A \cup \{(u, v)\} \subseteq T'$. Consequentemente, como T' é uma árvore geradora mínima, (u, v) é segura para A.

O Teorema 23.1 nos permite compreender melhor o funcionamento do método GENERIC-MST no grafo conexo $G = (V, E)$. À medida que o algoritmo progride, o conjunto A é sempre acíclico; caso contrário, uma árvore geradora mínima incluindo A conteria um ciclo, o que é uma contradição. Em qualquer ponto na execução, o grafo $G_A = (V, A)$ é uma floresta, e cada uma das componentes conexas de G_A é uma árvore. (Algumas das árvores podem conter apenas um vértice, como ocorre, por exemplo, quando o método começa: A é vazio e a floresta contém $|V|$ árvores, uma para

cada vértice.) Além disso, qualquer aresta segura (u, v) para A conecta componentes distintos de G_A, já que $A \cup \{(u, v)\}$ deve ser acíclico.

O laço **while** nas linhas 2-4 de GENERIC-MST é executado $|V| - 1$ vezes porque encontra uma das $|V| - 1$ arestas de uma árvore geradora mínima em cada iteração. No início, quando $A = 0/$, há $|V|$ árvores em G_A, e cada iteração reduz esse número em uma unidade. Quando a floresta contém apenas uma única árvore, o método termina.

Os dois algoritmos na Seção 23.2 utilizam o corolário do Teorema 23.1 apresentado a seguir.

Corolário 23.2

Seja $G = (V, E)$ um grafo conexo não dirigido com uma função peso de valor real w definida em E. Seja A um subconjunto de E que está incluído em alguma árvore geradora mínima para G, e seja $C = (V_C, E_C)$ uma componente conexa (árvore) na floresta $G_A = (V, A)$. Se (u, v) é uma aresta leve que conecta C a alguma outra componente em G_A, então (u, v) é segura para A.

Prova O corte $(V_C, V - V_C)$ respeita A, e (u, v) é então uma aresta leve para esse corte. Portanto, (u, v) é segura para A.

Exercícios

23.1-1 Seja (u, v) uma aresta de peso mínimo em um grafo conexo G. Mostre que (u, v) pertence a alguma árvore geradora mínima de G.

23.1-2 O professor Sabatier propõe a recíproca do Teorema 23.1 apresentada a seguir. Seja $G = (V, E)$ um grafo conexo não dirigido com uma função peso de valor real w definida em E. Seja A um subconjunto de E que está incluído em alguma árvore geradora mínima para G, seja $(S, V - S)$ qualquer corte de G que respeita A, e seja (u, v) uma aresta segura para A que cruza $(S, V - S)$. Então, (u, v) é uma aresta leve para o corte. Mostre que a hipótese do professor é incorreta, dando um contraexemplo.

23.1-3 Mostre que, se uma aresta (u, v) está contida em alguma árvore geradora mínima, então ela é uma aresta leve que cruza algum corte do grafo.

23.1-4 Dê um exemplo simples de um grafo conexo tal que no conjunto de arestas $\{(u, v)$: há um corte $(S, V - S)$ tal que (u, v) uma aresta leve que cruza $(S, V - S)\}$ não forma uma árvore geradora mínima.

23.1-5 Seja e uma aresta de peso máximo em algum ciclo do grafo conexo $G = (V, E)$. Prove que existe uma árvore geradora mínima de $G' = (V, E - \{e\})$ que também é uma árvore geradora mínima de G. Isto é, existe uma árvore geradora mínima de G que não inclui e.

23.1-6 Mostre que um grafo tem uma árvore geradora mínima única se, para todo corte do grafo, existe uma aresta leve única que cruza o corte. Mostre que a recíproca não é verdadeira, dando um contraexemplo.

23.1-7 Mostre que, se todos os pesos de arestas de um grafo são positivos, qualquer subconjunto de arestas que conecte todos os vértices e tenha peso total mínimo deve ser uma árvore. Dê um exemplo para mostrar que a mesma conclusão não decorre se permitimos que alguns pesos sejam não positivos.

23.1-8 Seja T uma árvore geradora mínima de um grafo G e seja L a lista ordenada dos pesos das arestas de T. Mostre que, para qualquer outra árvore geradora mínima T' de G, a lista L é também a lista ordenada de pesos de arestas de T'.

23.1-9 Seja T uma árvore geradora mínima de um grafo $G = (V, E)$ e seja V' um subconjunto de V. Seja T' o subgrafo de T induzido por V' e seja G' o subgrafo de G induzido por V'. Mostre que, se T' é conexo, então T' é uma árvore geradora mínima de G'.

23.1-10 Dado um grafo G e uma árvore geradora mínima T, suponha que diminuímos o peso de uma das arestas em T. Mostre que T ainda é uma árvore geradora mínima para G. Mais formalmente, seja T uma árvore geradora mínima para G com pesos de arestas dados pela função peso w. Escolha uma aresta $(x, y) \in T$ e um número positivo k, e defina a função peso w' por

$$w'(u,v) = \begin{cases} w(u,v) & \text{se } (u,v) \neq (x,y), \\ w(x,y) - k & \text{se } (u,v) = (x,y). \end{cases}$$

Mostre que T é uma árvore geradora mínima para G com pesos de arestas dados por w'.

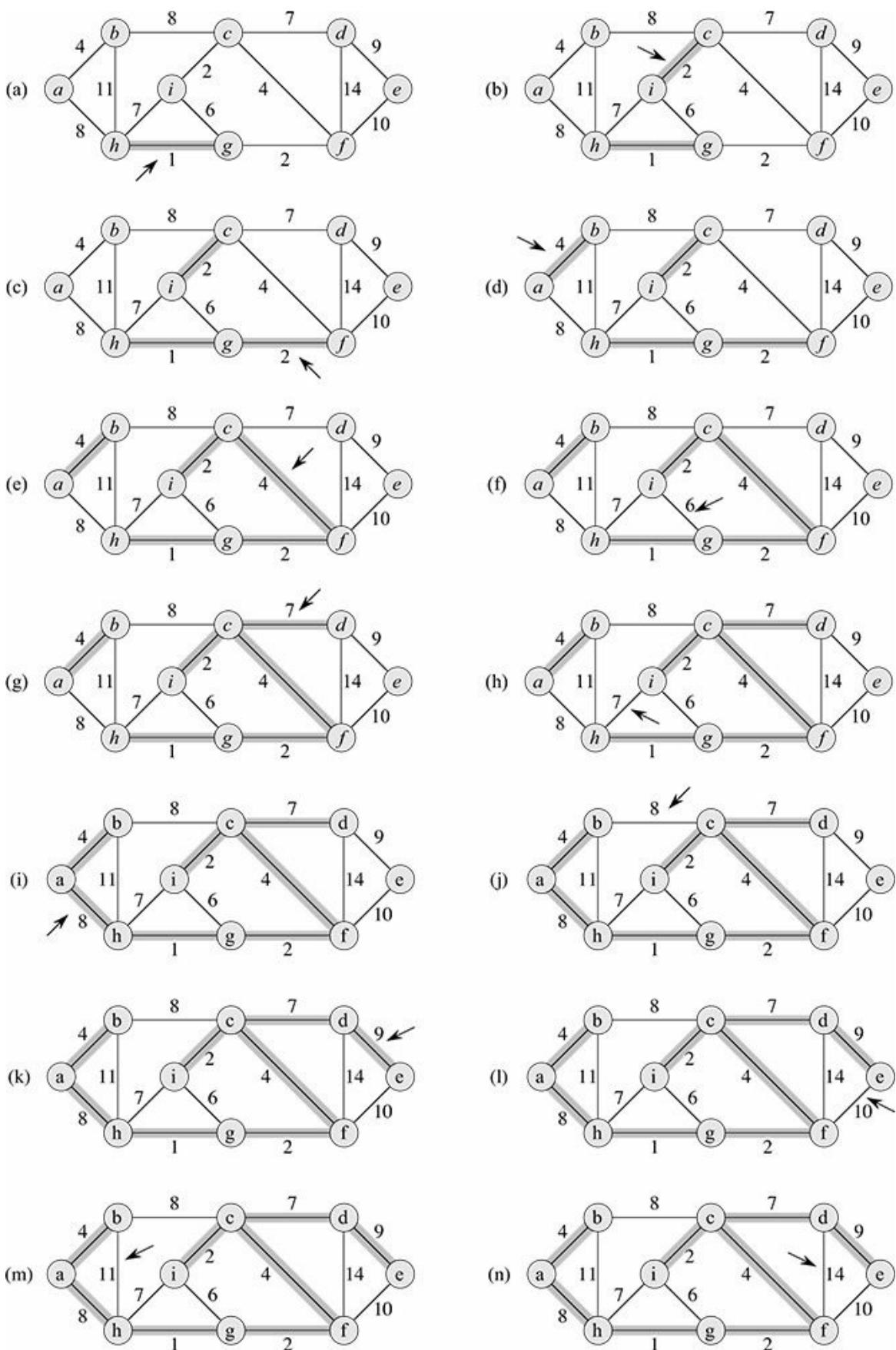
23.1-11 ★ Dado um grafo G e uma árvore geradora mínima T, suponha que diminuímos o peso de uma das arestas não presentes em T. Dê um algoritmo para encontrar a árvore geradora mínima no grafo modificado.

23.2 ALGORITMOS DE KRUSKAL E PRIM

Os dois algoritmos de árvore geradora mínima descritos nesta seção aperfeiçoam o método genérico. Cada um deles utiliza uma regra específica para determinar uma aresta segura na linha 3 de GENERIC-MST. No algoritmo de Kruskal, o conjunto A é uma floresta cujos vértices são todos os vértices do grafo dado. A aresta segura adicionada a A é sempre uma aresta de peso mínimo no grafo que conecta duas componentes distintas. No algoritmo de Prim, o conjunto A forma uma árvore única. A aresta segura adicionada a A é sempre uma aresta de peso mínimo que conecta a árvore a um vértice não presente na árvore.

Algoritmo de Kruskal

O algoritmo de Kruskal acha uma aresta segura para adicionar à floresta que está sendo desenvolvida encontrando, entre todas as arestas que conectam quaisquer duas árvores na floresta, uma aresta (u, v) de peso mínimo. Sejam C_1 e C_2 as duas árvores que são conectadas por (u, v). Visto que (u, v) deve ser uma aresta leve que conecta C_1 a alguma outra árvore, o Corolário 23.2 implica que (u, v) é uma aresta segura para C_1. O algoritmo de Kruskal se qualifica como um algoritmo guloso porque em cada etapa ele adiciona à floresta uma aresta de menor peso possível.

Nossa implementação do algoritmo de Kruskal é semelhante ao algoritmo para calcular componentes conexas da Seção 21.1. Utiliza uma estrutura de dados de conjuntos disjuntos para manter vários conjuntos disjuntos de elementos. Cada conjunto contém os vértices em uma árvore da floresta atual. A operação FIND-SET(u) retorna um elemento representativo do conjunto que contém u. Assim, podemos determinar se dois vértices u e v pertencem à mesma árvore testando se FIND-SET(u) é igual a FIND-SET(v). Para combinar as árvores, o algoritmo de Kruskal chama o procedimento UNION.

```
MST-KRUSKAL(G, w)
1   A = ∅
2   for cada vértice v ∈ G.V
3       MAKE-SET(v)
4   ordene as arestas de G.E em ordem não decrescente de peso w
5   for cada aresta (u, v) ∈ G.E, tomada em ordem não decrescente de peso
6       if FIND-SET(u) ≠ FIND-SET(v)
7           A = A ∪ {(u, v)}
8           UNION(u, v)
9   return A
```

A Figura 23.4 mostra como o algoritmo de Kruskal funciona. As linhas 1-3 inicializam o conjunto A para o conjunto vazio e criam $|V|$ árvores, cada uma contendo um vértice. O laço **for** das linhas 5-8 examina arestas em ordem de peso, do mais baixo ao mais alto. O laço verifica, para cada aresta (u, v), se os pontos extremos u e v pertencem à mesma árvore. Se pertencerem, então a aresta (u, v) não pode ser adicionada à floresta sem criar um ciclo, e a aresta será descartada. Caso contrário, os dois vértices pertencem a árvores diferentes. Nesse caso, a linha 7 adiciona a aresta (u, v) a A e a linha 8 intercala os vértices nas duas árvores.

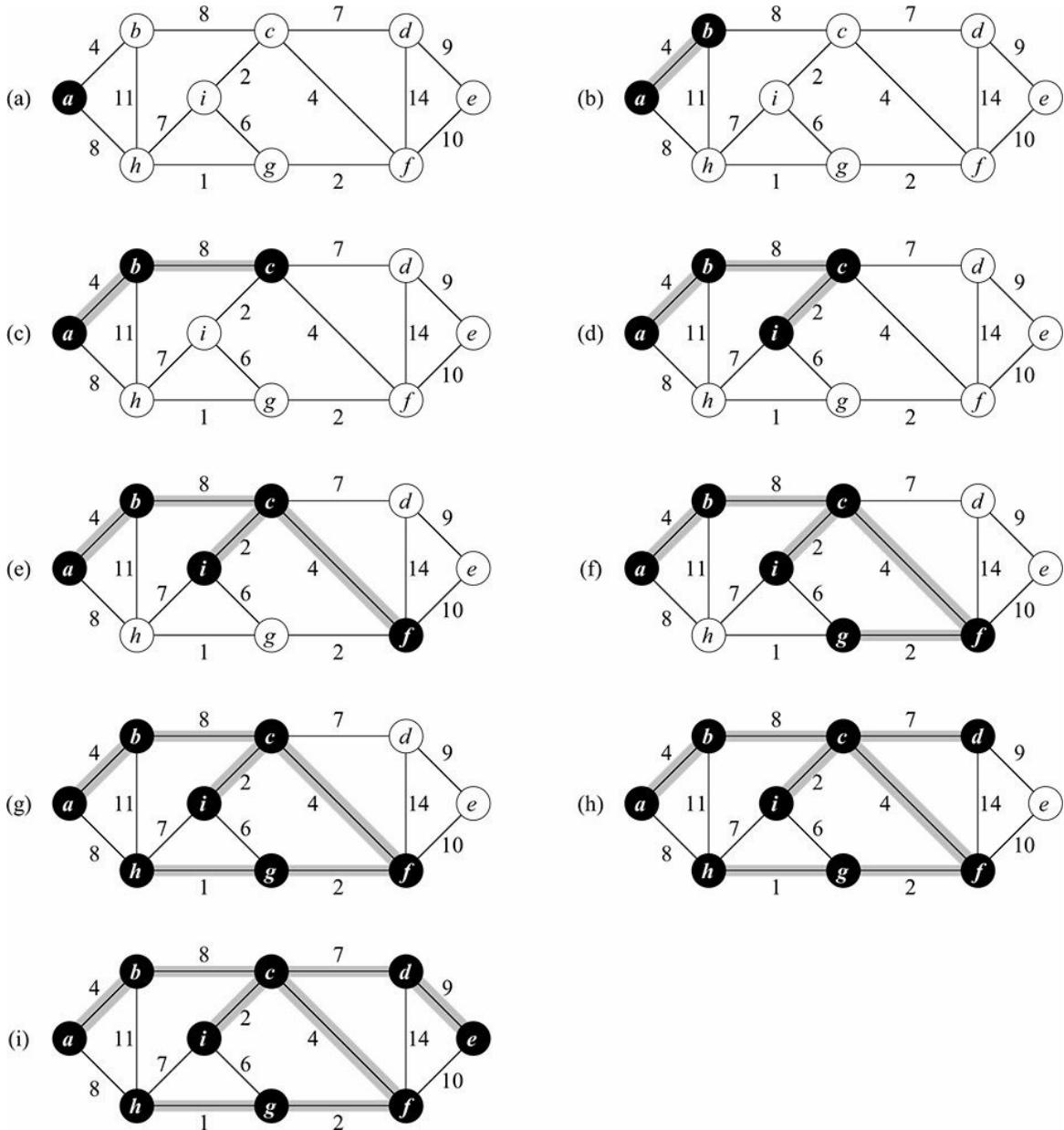
O tempo de execução do algoritmo de Kruskal para um grafo $G = (V, E)$ depende da implementação da estrutura de dados de conjuntos disjuntos. Suponhamos que usamos a implementação de floresta de conjuntos disjuntos da Seção 21.3 com as heurísticas de união por ordenação e compressão de caminho, já que essa é a implementação assintoticamente mais rápida conhecida. A inicialização do conjunto A na linha 1 demora o tempo $O(1)$, e o tempo para ordenar as arestas na linha 4 é $O(E \lg E)$. (Consideraremos em breve o custo das $|V|$ operações MAKE-SET no laço **for** das linhas 2-3.) O laço **for** das linhas 5-8 executa $O(E)$ operações FIND-SET e UNION na floresta de conjuntos disjuntos. Essas operações, mais as $|V|$ operações MAKE-SET, demoram o tempo total $O((V + E)\ \alpha(V))$, onde α é a função de crescimento muito lento definida na Seção 21.4. Como supomos que G é conexo, temos $|E| \geq |V| - 1$ e, assim, as operações de conjuntos disjuntos demoram o tempo $O(E\ \alpha(V))$. Além disso, visto que $\alpha(|V|) = O(\lg V) = O(\lg E)$, o tempo de execução total do algoritmo de Kruskal é $O(E \lg E)$. Observando que $|E| < |V|^2$, temos $\lg |E| = O(\lg V)$ e, portanto, podemos reescrever o tempo de execução do algoritmo de Kruskal como $O(E \lg V)$.

Figura 23.4 Execução do algoritmo de Kruskal no grafo da Figura 23.1. As arestas sombreadas pertencem à floresta que está sendo desenvolvida. O algoritmo considera cada aresta em sequência ordenada por peso. Uma seta aponta para a aresta que está sendo considerada em cada etapa do algoritmo. Se a aresta une duas árvores distintas na floresta, ela é adicionada à floresta, juntando assim as duas árvores.

Algoritmo de Prim

Como o algoritmo de Kruskal, o algoritmo de Prim é um caso especial do método genérico de árvore geradora mínima da Seção 23.1. O algoritmo de Prim funciona de modo muito semelhante ao algoritmo de Dijkstra para localizar caminhos mínimos em um grafo, que veremos na Seção 24.3. O algoritmo de Prim tem a seguinte propriedade: as arestas no conjunto A sempre formam uma árvore única. Como mostra a Figura 23.5, a árvore começa em um vértice raiz arbitrário r e aumenta até que a árvore abranja todos os vértices em V. Cada etapa adiciona à árvore A uma aresta leve que conecta A a um vértice isolado — um vértice no qual nenhuma aresta de A incide. Pelo Corolário 23.2, essa regra adiciona apenas arestas que são seguras para A; portanto, quando o algoritmo termina, as arestas em A formam uma árvore geradora mínima. Essa estratégia se qualifica como gulosa, já que a cada etapa ela adiciona à árvore uma aresta que contribui com a mínima quantidade possível para o peso da árvore.

Para implementar o algoritmo de Prim eficientemente, precisamos de um modo rápido para selecionar uma nova aresta a ser adicionada à árvore formada pelas arestas em A. No pseudocódigo a seguir, o grafo conexo G e a raiz r da árvore geradora mínima a ser desenvolvida são entradas para o algoritmo. Durante a execução do algoritmo, todos os vértices que *não* estão na árvore residem em uma fila de prioridade mínima Q baseada em um atributo *chave*. Para cada vértice v, o atributo $v.chave$ é o peso mínimo de qualquer aresta que conecta v a um vértice na árvore; por convenção, $v.chave = \infty$ se não existe nenhuma aresta desse tipo. O atributo $v.p$ nomeia o pai de v na árvore. O algoritmo mantém implicitamente o conjunto A de GENERIC-MST como

$$A = \{(v, v.p) : v \in V - \{r\} - Q\} .$$

Quando o algoritmo termina, a fila de prioridade mínima Q está vazia; portanto, a árvore geradora mínima A para G é

$$A = \{(v, v.p) : v \in V - \{r\}\} .$$ MST-PRIM(G, w, r)

```
MST-PRIM(G, w, r)
1     for cada u ∈ V[G]
2           u.chave = ∞
3           u.π = NIL
4     r.chave = 0
5     Q = V[G]
6     while Q ≠ Ø
7           u = EXTRACT-MIN(Q)
8           for cada v ∈ G. Adj[u]
9                 if v ∈ Q e w(u, v) v.chave
10                      v.π = u
11                      v.chave = w(u, v)
```

A Figura 23.5 mostra como o algoritmo de Prim funciona. As linhas 1-5 definem a chave de cada vértice como ∞ (exceto a raiz r, cuja chave é definida como 0 e, por isso, será o primeiro vértice processado), define o pai de cada vértice como NIL e inicializa a fila de prioridade mínima Q para conter todos os vértices. O algoritmo mantém o seguinte invariante de laço de três partes:

Figura 23.5 Execução do algoritmo de Prim no grafo da Figura 23.1. O vértice raiz é a. Arestas sombreadas estão na árvore que está sendo desenvolvida, e os vértices pretos estão na árvore. Em cada etapa do algoritmo, os vértices na árvore determinam um corte do grafo, e uma aresta leve que cruza o corte é acrescentada à árvore. Na segunda etapa, por exemplo, o algoritmo tem a opção de adicionar a aresta (b, c) ou a aresta (a, h) à árvore, visto que ambas são arestas leves que cruzam o corte.

Antes de cada iteração do laço **while** das linhas 6-11,

1. $A = (v, v.p) : v \in V - \{r\} - Q\}$.
2. Os vértices que já estão presentes na árvore geradora mínima são aqueles em $V - Q$.
3. Para todos os vértices $v \in Q$, se $v.p \neq$ NIL, então $v.chave < \infty$ e $v.chave$ é o peso de uma aresta leve $(v, v.p)$ que conecta v a algum vértice já inserido na árvore geradora mínima.

A linha 7 identifica um vértice $u \in Q$ incidente em uma aresta leve que cruza o corte $(V - Q, Q)$ (com exceção da primeira iteração, na qual $u = r$ devido à linha 4). Remover u do conjunto Q o acrescenta ao conjunto $V - Q$ de vértices na árvore, adicionando assim $(u, u.p)$ a A. O laço **for** das linhas 8-11 atualiza os atributos *chave* e p de cada vértice v adjacente a u, mas não na árvore; por consequência, mantém a terceira parte do invariante de laço.

O tempo de execução do algoritmo de Prim depende de como implementamos a fila de prioridades Q. Se implementamos Q como um heap mínimo binário (veja o Capítulo 6), podemos usar o procedimento BUILD-MIN-HEAP para executar as linhas 1-5 no tempo $O(V)$. O corpo do laço **while** é executado $|V|$ vezes e, como cada operação de EXTRACT-MIN demora o tempo $O(\lg V)$, o tempo total para todas as chamadas a EXTRACT-MIN é $O(V \lg V)$. O laço **for** nas linhas 8–11 é executado $O(E)$ vezes no total, já que a soma dos comprimentos de todas as listas de adjacências é $2|E|$. Dentro do laço **for**, podemos implementar o teste de pertinência em Q na linha 9 em tempo constante mantendo um bit para cada vértice que informa se ele está ou não em Q e atualizando o bit quando o vértice é removido de Q. A atribuição na linha 11 envolve uma operação DECREASE-KEY implícita no heap mínimo, que um heap binário mínimo suporta no tempo $O(\lg V)$. Assim, o tempo total para o algoritmo de Prim é $O(V \lg V + E \lg V) = O(E \lg V)$, que é assintoticamente igual ao da nossa implementação do algoritmo de Kruskal.

Podemos melhorar o tempo de execução assintótico do algoritmo de Prim usando heaps de Fibonacci. O Capítulo 19 mostra que, se um heap de Fibonacci contém $|V|$ elementos, uma operação EXTRACT-MIN leva o tempo amortizado $O(\lg V)$ e uma operação DECREASE-KEY (para implementar a linha 11) leva o tempo amortizado $O(1)$. Então, se usarmos um heap de Fibonacci para implementar a fila de prioridade mínima Q, o tempo de execução do algoritmo de Prim melhorará para $O(E + V \lg V)$.

Exercícios

23.2-1 O algoritmo de Kruskal pode devolver diferentes árvores geradoras para o mesmo grafo de entrada G, dependendo de como as ligações são rompidas quando as arestas são ordenadas. Mostre que, para cada árvore geradora mínima T de G, existe um modo de ordenar as arestas de G no algoritmo de Kruskal, de tal forma que o algoritmo retorne T.

23.2-2 Suponha que representamos o grafo $G = (V, E)$ como uma matriz de adjacências. Dê uma implementação simples do algoritmo de Prim para esse caso que seja executada no tempo $O(V_2)$.

23.2-3 Para um grafo esparso $G = (V, E)$, onde $|E| = Q(V)$, a implementação do algoritmo de Prim com um heap de Fibonacci é assintoticamente mais rápida que a implementação de heap binário? E para um grafo denso, onde $|E| = Q(V_2)$? Como os tamanhos $|E|$ e $|V|$ devem estar relacionados para que a implementação de heap de Fibonacci seja assintoticamente mais rápida que a implementação de heap binário?

23.2-4 Suponha que todos os pesos de arestas em um grafo sejam inteiros na faixa de 1 a $|V|$. Com que rapidez é possível executar o algoritmo de Kruskal? E se os pesos de arestas forem inteiros na faixa de 1 a W para alguma constante W?

23.2-5 Suponha que todos os pesos de arestas em um grafo sejam inteiros na faixa de 1 a $|V|$. Com que rapidez é possível executar o algoritmo de Prim? E se os pesos de arestas forem inteiros na faixa de 1 a W para alguma constante W?

23.2-6* Suponha que os pesos de arestas em um grafo estejam uniformemente distribuídos no intervalo meio aberto [0, 1). Qual algoritmo, o de Kruskal ou o de Prim, pode tornar a execução mais rápida?

23.2-7* Suponha que um grafo G tenha uma árvore geradora mínima já calculada. Com que rapidez podemos atualizar a árvore geradora mínima se adicionarmos a G um novo vértice e arestas incidentes?

23.2-8 O professor Borden propõe um novo algoritmo de divisão e conquista para calcular árvores geradoras mínimas, que apresentamos a seguir. Dado um grafo $G = (V, E)$, particione o conjunto V de vértices em dois conjuntos V_1 e V_2, tais que a diferença entre $|V_1|$ e $|V_2|$ seja no máximo 1. Seja E_1 o conjunto de arestas incidentes somente em vértices de V_1 e seja E_2 o conjunto de arestas incidentes somente em vértices de V_2.

Resolva recursivamente um problema de árvore geradora mínima para cada um dos dois subgrafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$. Finalmente, selecione a aresta de peso mínimo em E que cruza o corte (V_1, V_2) e use essa aresta para unir as duas árvores geradoras mínimas resultantes em uma única árvore geradora. Demonstre que o algoritmo calcula corretamente uma árvore geradora mínima de G ou dê um exemplo para o qual o algoritmo não funciona.

Problemas

23-1 *Segunda melhor árvore geradora mínima*

Seja $G = (V, E)$ um grafo conexo não dirigido com função peso $w : E \rightarrow$ e suponha que $|E| \geq |V|$ e todos os pesos de arestas sejam distintos.

Definimos uma segunda melhor árvore geradora mínima da seguinte maneira: seja o conjunto de todas as árvores de G, e seja T' uma árvore geradora mínima de G. Então, uma ***segunda melhor*** árvore geradora mínima é uma árvore geradora T tal que $w(T) = \min_{T'' \in - \{T'\}} \{w(T'')\}$.

a. Mostre que a árvore geradora mínima é única, mas que a segunda melhor árvore geradora mínima não precisa ser única.

b. Seja T uma árvore geradora mínima de G. Prove que G contém arestas $(u, v) \in T$ e $(x, y) \notin T$ tais que $T - \{(u, v)\} \cup \{(x, y)\}$ é uma segunda melhor árvore geradora mínima de G.

c. Seja T uma árvore geradora de G e, para quaisquer dois vértices $u, v \in V$, seja $max u, v$ uma aresta de peso máximo no caminho simples único entre u e v em T. Descreva um algoritmo de tempo $O(V^2)$ que, dado T, calcule $max u, v$ para todo $u, v \in V$.

d. Dê um algoritmo eficiente para calcular a segunda melhor árvore geradora mínima de G.

23-2 *Árvore* **geradora** *mínima em grafos esparsos*

Para um grafo conexo muito esparso $G = (V, E)$, podemos melhorar ainda mais o tempo de execução $O(E + V \lg V)$ do algoritmo de Prim com heaps de Fibonacci, executando um pré-processamento de G para diminuir o número de vértices antes da execução do algoritmo de Prim. Em particular, escolhemos, para cada vértice u, a aresta de peso mínimo (u, v) incidente em u e colocamos (u, v) na árvore geradora mínima em construção. Depois, contraímos todas as arestas escolhidas (veja a Seção B.4). Em vez de contrair essas arestas uma por vez, primeiro identificamos conjuntos de vértices que estão unidos no mesmo novo vértice. Então, criamos o grafo que teria resultado da contração dessas arestas uma por vez, mas fazemos isso "renomeando" arestas de acordo com os conjuntos nos quais seus pontos extremos foram colocados. Várias arestas do grafo original podem ser renomeadas como iguais a outras. Em tal caso, resulta somente uma aresta, e seu peso é o mínimo entre os pesos das arestas originais correspondentes.

Inicialmente, definimos a árvore geradora mínima T que está sendo construída para ser vazia e, para cada aresta $(u, v) \in E$, inicializamos os atributos u, v $orig = (u, v)$ e u, v $c = w(u, v)$. Usamos o atributo *orig* para referenciar a aresta do grafo inicial que está associada a uma aresta no grafo contraído. O atributo c contém o peso de uma aresta e, à medida que as arestas são contraídas, nós o atualizamos de acordo com o esquema descrito para escolha de pesos de arestas. O procedimento MST-REDUCE toma as entradas G e T e retorna um grafo contraído G' com atributos atualizados *orig'* e c'. O procedimento também acumula arestas de G na árvore geradora mínima T. MST-REDUCE(G, T)

```
MST-REDUCE(G, T)
1       for cada v ∈ G.V
2               v.mark = FALSE
3               MAKE-SET(v)
4       for cada u ∈ G.V
5               if u.mark == FALSE
6                       escolher v ∈ G.Adj[u] tal que (u, v).c é minimizado
7                       UNION(u, v)
8                       T = T ∪ {(u, v).orig}
9                       u.mark = v.mark = TRUE
10      G′.V = {FIND-SET(v) : v ∈ G.V}
11      G′.E = ∅
12      for cada (x, y) ∈ G.E
13              u = FIND-SET(x)
14              v = FIND-SET(y)
15              if (u, v) ∉ G′.E
16                      G′.E = G′.E ∪ {(u, v)}
17                      (u, v).orig′ = (x, y).orig
18                      (u, v).c′ = (x, y).c′
19              else if (x, y).c < (u, v).c′
20                              (u, v).orig′ = (x, y).orig
21                              (u, v).c′ = (x, y).c
22      construa listas de adjacências G′.Adj para G′
23      return G′ e T
```

a. Seja T o conjunto de arestas devolvidas por MST-REDUCE e seja A a árvore geradora mínima do grafo G' formada pela chamada MST-PRIM(G', c', r), onde c' é o atributo peso para as arestas de $G'.E$ e r é qualquer vértice em $G'.V$. Prove que $T \cup \{x, y \quad orig' : (x, y) \in A\}$ é uma árvore geradora mínima de G.

b. Demonstre que $|G'.V| \leq |V|/2$.

c. Mostre como implementar MST-REDUCE de modo que ele seja executado no tempo $O(E)$. (*Sugestão*: Utilize estruturas de dados simples.)

d. Suponha que executamos k fases de MST-REDUCE usando a saída G' produzida por uma fase como a entrada de G para a fase seguinte e acumulando arestas em T. Demonstre que o tempo de execução global das k fases é $O(kE)$.

e. Suponha que, depois de executar k fases de MST-REDUCE, como na parte (d), executamos o algoritmo de Prim chamando MST-PRIM(G', c', r), onde G', com atributo peso c', é retornado pela última fase e r é qualquer vértice em $G'.V$. Mostre como escolher k de modo que o tempo de execução global seja $O(E \lg \lg V)$. Demonstre que sua escolha de k minimiza o tempo de execução assintótico global.

f. Para quais valores de $|E|$ (em termos de $|V|$) o algoritmo de Prim com pré-processamento é superior assintoticamente ao algoritmo de Prim sem pré-processamento?

23-3 Árvore geradora de gargalo

Uma ***árvore geradora de gargalo*** T de um grafo não dirigido G é uma árvore geradora de G cujo maior peso de aresta é mínimo em relação a todas as árvores geradoras de G. Dizemos que o valor da árvore geradora de gargalo é o peso da aresta de peso máximo em T.

a. Demonstre que uma árvore geradora mínima é uma árvore geradora de gargalo.

A parte (a) mostra que encontrar uma árvore geradora de gargalo não é mais difícil que encontrar uma árvore geradora mínima. Nas partes restantes, mostraremos que é possível encontrá-la em tempo linear.

b. Dê um algoritmo de tempo linear que, dado um grafo G e um inteiro b, determina se o valor da árvore geradora de gargalo é no máximo b.

c. Use seu algoritmo para a parte (b) como uma sub-rotina em um algoritmo de tempo linear para o problema da árvore geradora de gargalo. (*Sugestão:* Seria interessante você usar uma sub-rotina que contraia conjuntos de arestas, como no procedimento MST-REDUCE descrito no Problema 23-2.)

23-4 *Algoritmos alternativos de árvore geradora mínima*

Neste problema, apresentamos pseudocódigo para três algoritmos diferentes. Cada um toma um grafo conexo e uma função peso como entrada e retorna um conjunto de arestas T. Para cada algoritmo, prove que T é uma árvore geradora mínima ou que T não é necessariamente uma árvore geradora mínima. Descreva também a implementação mais eficiente de cada algoritmo, quer ele calcule ou não uma árvore geradora mínima.

a. MAYBE-MST-A(G, w)

1 ordenar as arestas em ordem não crescente de pesos de arestas w
2 $T = E$
3 **for** cada aresta e, tomada em ordem não crescente de peso
4 **if** $T - \{e\}$ é um grafo conexo
5   $T = T - e$
6 **return** T

b. MAYBE-MST-B(G, w)

1 $T = \emptyset$
2 **for** cada aresta e, tomada em ordem arbitrária
3 **if** $T \cup \{e\}$ não tem nenhum ciclo
4   $T = T \cup \{e\}$
5 **return** T

c. MAYBE-MST-C(G, w)

1 $T = \emptyset$
2 **for** cada aresta e, tomada em ordem arbitrária
3 $T = T \cup \{e\}$
4 **if** T tem um ciclo c
5   seja e' uma aresta de peso máximo em c
6   $T = T - \{e'\}$
7 **return** T

NOTAS DO CAPÍTULO

Tarjan [330] faz um levantamento do problema da árvore geradora mínima e dá excelente material avançado. Graham e Hell [151] compilaram um histórico do problema da árvore geradora mínima.

Tarjan atribui o primeiro algoritmo de árvore geradora mínima a um artigo de 1926 produzido por O. Boruvka. O algoritmo de Boruvka consiste na execução de $O(\lg V)$ iterações do procedimento MST-REDUCE descrito no Problema 23-2. O algoritmo de Kruskal foi apresentado por Kruskal [222] em 1956. O algoritmo comumente conhecido como algoritmo de Prim foi de fato desenvolvido por Prim [285], mas também foi criado antes por V. Jarník em 1930.

A razão por que algoritmos gulosos são efetivos para encontrar árvores geradoras mínima é que o conjunto de florestas de um grafo forma um matroide gráfico (veja a Seção 16.4).

Quando $|E| = (V \lg V)$, o algoritmo de Prim implementado com heaps de Fibonacci é executado no tempo $O(E)$. No caso de grafos mais esparsos, usando uma combinação das ideias do algoritmo de Prim, do algoritmo de Kruskal e do algoritmo de Boruvka, juntamente com estruturas de dados avançadas, Fredman e Tarjan [114] dão um algoritmo que é executado no tempo $O(E \lg^* V)$. Gabow, Galil, Spencer e Tarjan [120] melhoraram esse algoritmo para ser executado no tempo $O(E \lg \lg^* V)$. Chazelle [60] dá um algoritmo que é executado no tempo $O(E\, \hat{\alpha}(E, V))$, onde $\hat{\alpha}(E, V)$ é a inversa funcional da função de Ackermann (veja nas notas do Capítulo 21 uma breve discussão da função de Ackermann e sua inversa). Diferentemente dos algoritmos de árvore geradora mínima anteriores, o algoritmo de Chazelle não segue o método guloso.

Um problema relacionado é a ***verificação de árvores geradora***, na qual temos um grafo $G = (V, E)$ e uma árvore $T \subseteq E$ e desejamos determinar se T é uma árvore geradora mínima de G. King [203] dá um algoritmo de tempo linear para verificação de árvores geradoras, fundamentado no trabalho anterior de Komlós [215] e Dixon, Rauch e Tarjan [91].

Os algoritmos que citamos aqui são todos determinísticos e se enquadram no modelo baseado em comparação descrito no Capítulo 8. Karger, Klein e Tarjan [195] dão um algoritmo aleatorizado de árvore geradora mínima que é executado no tempo esperado $O(V + E)$. Esse algoritmo emprega recursão de um modo semelhante ao algoritmo de seleção de tempo linear na Seção 9.3: uma chamada recursiva para um problema auxiliar identifica um subconjunto das arestas E' que não podem estar em nenhuma árvore geradora mínima. Então, uma outra chamada recursiva em $E - E'$ encontra a árvore geradora mínima. O algoritmo também usa ideias do algoritmo de Boruvka e do algoritmo de King para verificação de árvores geradoras.

Fredman e Willard [116] mostraram como encontrar uma árvore geradora mínima no tempo $O(V + E)$ usando um algoritmo determinístico que não é baseado em comparação. Seu algoritmo supoem que os dados são inteiros de b bits e que a memória do computador consiste em palavras endereçáveis de b bits.

---

[1]A expressão "árvore geradora mínima" é uma forma abreviada da expressão "árvore geradora de peso mínimo". Não estamos, por exemplo, minimizando o número de arestas em T, já que todas as árvores geradoras têm exatamente $|V| - 1$ arestas de acordo com o Teorema B.2.

24 CAMINHOS MÍNIMOS DE FONTE ÚNICA

Um motorista deseja encontrar a rota mais curta possível do Rio de Janeiro a São Paulo. Dado um mapa rodoviário do Brasil no qual a distância entre cada par de interseções adjacentes esteja marcada, como ele pode determinar essa rota mais curta?

Um modo possível seria enumerar todas as rotas do Rio de Janeiro a São Paulo, somar as distâncias em cada rota e selecionar a mais curta. Porém, é fácil ver que até mesmo se deixarmos de lado as rotas que contêm ciclos, o motorista teria de examinar um número enorme de possibilidades, a maioria das quais simplesmente não valeria a pena considerar. Por exemplo, uma rota do Rio de Janeiro a São Paulo passando por Brasília é sem dúvida uma escolha ruim porque Brasília está várias centenas de quilômetros fora do caminho.

Neste capítulo e no Capítulo 25, mostraremos como resolver tais problemas eficientemente. Em um ***problema de caminhos mínimos***, temos um grafo dirigido ponderado $G = (V, E)$, com função peso $w : E \rightarrow$ que mapeia arestas para pesos de valores reais. O ***peso*** do caminho $p = \langle v_0, v_1, ..., v_k \rangle$ é a soma dos pesos de suas arestas constituintes:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Definimos o ***peso do caminho mínimo*** de u a v por

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{Se há um caminho de } u \text{ para } v, \\ \infty & \text{caso contrário} \end{cases}$$

Então, um ***caminho mínimo*** do vértice u ao vértice v é definido como qualquer caminho p com peso $w(p) = d(u, v)$.

No exemplo da rota entre o Rio de Janeiro e São Paulo, podemos modelar o mapa rodoviário como um grafo: vértices representam interseções, arestas representam segmentos de estradas entre interseções e pesos de arestas representam distâncias rodoviárias. Nossa meta é encontrar um caminho mínimo de um dado entroncamento de rodovias no Rio de Janeiro a um dado entroncamento de rodovias em São Paulo.

Pesos de arestas podem representar outras medidas que não sejam distâncias, como tempo, custo, multas, prejuízos ou qualquer outra quantidade que se acumule linearmente ao longo de um caminho e que seria interessante minimizar.

O algoritmo de busca em largura da Seção 22.2 é um algoritmo de caminhos mínimos que funciona em grafos não ponderados, isto é, grafos nos quais cada aresta tem peso unitário. Como muitos dos conceitos da busca em largura surgem no estudo de caminhos mínimos em grafos ponderados, seria interessante o leitor revisar a Seção 22.2 antes de continuar.

Variantes

Neste capítulo, focalizaremos o ***problema de caminhos mínimos de fonte única***: dado um grafo $G = (V, E)$, queremos encontrar um caminho mínimo de determinado vértice de ***origem*** $s \in V$ a todo vértice $v \in V$. O algoritmo para o problema da fonte única pode resolver muitos outros problemas, entre os quais as variantes apresentadas a seguir.

Problema de caminhos mínimos com um só destino: Encontrar um caminho mínimo até um determinado vértice de ***destino*** t a partir de cada vértice v. Invertendo a direção de cada aresta no grafo, podemos reduzir esse problema a um problema de fonte única.

Problema do caminho mínimo para um par: Encontrar um caminho mínimo de u a v para vértices u e v dados. Se resolvermos o problema de fonte única com vértice de fonte u, também resolveremos esse problema. Além disso, todos os algoritmos conhecidos para esse problema têm o mesmo tempo de execução assintótica do pior caso que os melhores algoritmos de fonte única.

Problema de caminhos mínimos para todos os pares: Encontrar um caminho mínimo de u a v para todo par de vértices u e v. Embora seja possível resolver esse problema executando um algoritmo de fonte única uma vez para cada vértice, em geral podemos resolvê–lo mais rapidamente. Além disso, sua estrutura é interessante por si só. O Capítulo 25 estuda em detalhes o problema para todos os pares.

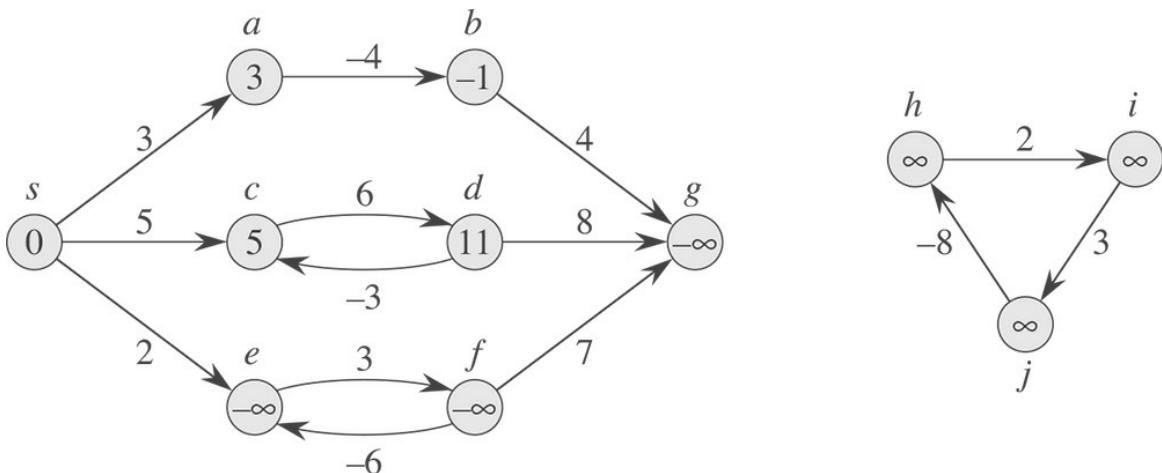
Subestrutura ótima de um caminho mínimo

Em geral, algoritmos de caminhos mínimos se baseiam na seguinte propriedade: um caminho mínimo entre dois vértices contém outros caminhos mínimos. (O algoritmo de fluxo máximo de EdmondsKarp no Capítulo 26 também se baseia nessa propriedade.) Lembrese de que subestrutura ótima é um dos indicadores fundamentais da possível aplicabilidade da programação dinâmica (Capítulo 15) e do método guloso (Capítulo 16). O algoritmo de Dijkstra, que veremos na Seção 24.3, é um algoritmo guloso, e o algoritmo de FloydWarshall, que encontra caminhos mínimos entre todos os pares de vértices (veja a Seção 25.2), é um algoritmo de programação dinâmica. O lema a seguir enuncia com maior exatidão a propriedade de subestrutura ótima de caminhos mínimos.

Lema 24.1 (Subcaminhos de caminhos mínimos são caminhos mínimos)

Dado um grafo dirigido ponderado $G = (V, E)$ com função peso $w : E \rightarrow$, seja $p = \langle v_1, v_2, ..., v_k \rangle$ um caminho mínimo do vértice v_0 ao vértice v_k e, para quaisquer i e j tais que $1 \leq i \leq j \leq k$, seja $p_{ij} = \langle v_i, v_i + 1, ..., v_j \rangle$ o subcaminho p do vértice v_i ao vértice v_j. Então, p_{ij} é um caminho mínimo de v_i a v_j.

Prova Se decompusermos caminho $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$, teremos que $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Agora, suponha que exista um caminho p'_{ij} de v_i a v_j com peso $w(p'_{ij}) < w(p_{ij})$. Então, $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ é um caminho de v_0 a v_k cujo peso $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ é menor que $w(p)$, o que contradiz a hipótese de que p seja um caminho mínimo de v_0 a v_k. ■

Arestas de peso negativo

Algumas instâncias do problema de caminhos mínimos de fonte única podem incluir arestas cujos pesos são negativos. Se o grafo $G = (V, E)$ não contém nenhum ciclo de peso negativo que possa ser alcançado da fonte s, então para todo $v \in V$, o peso do caminho mínimo $d(s, v)$ permanece bem definido, mesmo que tenha um valor negativo. Contudo, se o grafo contém um ciclo de peso negativo que possa ser alcançado a partir de s, os pesos de caminhos mínimos não são bem definidos. Nenhum caminho de s a um vértice no ciclo pode ser um caminho mínimo — sempre podemos encontrar um caminho de peso menor seguindo o caminho "mínimo" proposto e depois percorrendo o ciclo de peso negativo. Se houver um ciclo de peso negativo em algum caminho de s a v, definiremos $d(s, v) = -\infty$

A Figura 24.1 ilustra o efeito de pesos negativos e ciclos de pesos negativos em pesos de caminhos mínimos. Como há somente um caminho de s a a (o caminho $\langle s, a\rangle$), temos $d(s, a) = w(s, a) = 3$. De modo semelhante, há somente um caminho de s a b, e assim $d(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. Há um número infinito de caminhos de s a c: $\langle s, c\rangle$, $\langle s, c, d, c\rangle$, $\langle s, c, d, c, d, c\rangle$, e assim por diante. Como o ciclo $\langle c, d, c\rangle$, tem peso $6 + (-3) = 3 > 0$, o caminho mínimo de s a c é $\langle s, c\rangle$, com peso $d(s, c) = w(s, c) = 5$. Do mesmo modo, o caminho mínimo de s a d é $\langle s, c, d\rangle$, com peso $d(s, d) = w(s, c) + w(c, d) = 11$. Analogamente, há um número infinito de caminhos de s a e: $\langle s, e\rangle$, $\langle s, e, f, e\rangle$, $\langle s, e, f, e, f, e\rangle$, e assim por diante. Porém, visto que o ciclo $\langle e, f, e\rangle$, tem peso $3 + (-6) = -3 < 0$, não há nenhum caminho mínimo de s a e. Percorrendo o ciclo de peso negativo $\langle e, f, e\rangle$ um número arbitrário de vezes, podemos encontrar caminhos de s a e com pesos negativos arbitrariamente grandes e, assim, $d(s, e) = -\infty$. De modo semelhante, $d(s, f) = -\infty$. Como g pode ser alcançado de f, também podemos encontrar caminhos com pesos negativos arbitrariamente grandes de s a g e, portanto, $d(s, g) = -\infty$. Os vértices h, i e j também formam um ciclo de peso negativo. Contudo, eles não podem ser alcançados de s e, assim, $d(s, h) = d(s, i) = d(s, j) = \infty$.

Alguns algoritmos de caminhos mínimos, como o algoritmo de Dijkstra, consideram que todos os pesos de arestas no grafo de entrada são não negativos, como no exemplo do mapa rodoviário. Outros, como o algoritmo de Bellman–Ford, permitem arestas de peso negativo no grafo de entrada e produzem uma resposta correta desde que nenhum ciclo de peso negativo possa ser alcançado da fonte. Normalmente, se houver tal ciclo de peso negativo, o algoritmo poderá detectar e relatar sua existência.

Ciclos

Um caminho mínimo pode conter um ciclo? Como acabamos de ver, ele não pode conter um ciclo de peso negativo. Nem pode conter um ciclo de peso positivo, já que remover o ciclo do caminho produz um caminho com os mesmos vértices de fonte e destino, e um peso de caminho mais baixo. Isto é, se $p' = \langle v_0, v_1, ..., v_k\rangle$ é um caminho e $c = \langle v_i, v_i + 1, ..., v_j\rangle$ é um ciclo de peso positivo nesse caminho (de modo que $v_i = v_j$ e $w(c) > 0$), então o caminho $p' = \langle v_0, v_1, ..., v_i, v_j + 1, v_j + 2, ..., v_k\rangle$ tem peso $w(p') = w(p) - w(c) < w(p)$ e, portanto, p não pode ser um caminho mínimo de v_0 a v_k.

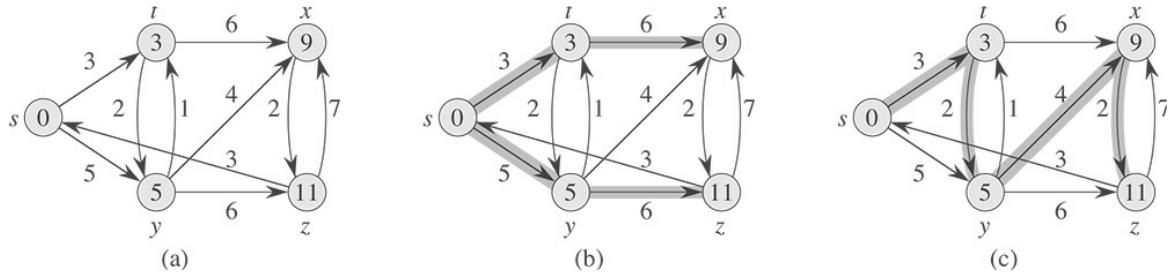
Figura 24.1 Pesos negativos de arestas em um grafo dirigido. O peso do caminho mínimo que sai da fonte s aparece dentro de cada vértice. Como os vértices e e f formam um ciclo de peso negativo que pode ser alcançado de s, os pesos de seus caminhos mínimos são $-\infty$. Como o vértice g pode ser alcançado de um vértice cujo peso de caminho mínimo é $-\infty$, também ele tem um peso de caminho mínimo $-\infty$. Vértices como h, i e j não podem ser alcançados de s e, assim, seus pesos de caminho mínimo são ∞, embora eles se encontrem em um ciclo de peso negativo.

Isso deixa apenas ciclos de peso 0. Podemos remover um ciclo de peso 0 de qualquer caminho para produzir um outro caminho cujo peso é o mesmo. Assim, se existe um caminho mínimo de um vértice de fonte *s* a um vértice de destino *v* que contém um ciclo de peso 0, então existe outro caminho mínimo de *s* a *v* sem esse ciclo. Enquanto que um caminho mínimo tenha ciclos de peso 0, podemos remover repetidamente esses ciclos do caminho até que tenhamos um caminho mínimo livre de ciclos. Então, sem perda da generalidade, podemos considerar que, quando estamos encontrando caminhos mínimos, eles não têm nenhum ciclo, isto é, são caminhos simples. Visto que qualquer caminho acíclico em um grafo $G = (V, E)$ contém no máximo $|V|$ vértices distintos, ele também contém no máximo $|V| - 1$ arestas. Assim, podemos restringir nossa atenção a caminhos mínimos que tenham no máximo $|V| - 1$ arestas.

Representação de caminhos mínimos

Muitas vezes, desejamos calcular não apenas pesos de caminhos mínimos, mas também os vértices nos caminhos mínimos. A representação que usamos para caminhos mínimos é semelhante à que utilizamos para árvores em largura na Seção 22.2. Dado um grafo $G = (V, E)$, mantemos para cada vértice $v \in V$ um ***predecessor*** $v.p$ que é um outro vértice ou NIL. Os algoritmos de caminhos mínimos apresentados neste capítulo definem os atributos p de modo que a cadeia de predecessores que se origina em um vértice v percorra um caminho mínimo de s a v em sentido contrário. Assim, dado um vértice v para o qual $v.p \neq$ NIL o procedimento PRINT-PATH(G, s, v) da Seção 22.2 imprimirá um caminho mínimo de s a v.

Entretanto, durante a execução de um algoritmo de caminhos mínimos, os valores p poderiam não indicar caminhos mínimos. Como na busca em largura, estaremos interessados no ***subgrafo dos predecessores*** $G_p = (V_p, E_p)$ induzido pelos valores p. Aqui, novamente definimos o conjunto de vértices V_p como o conjunto de vértices de G com predecessores não NIL mais a fonte s:

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}.$$

O conjunto de arestas dirigidas E_p é o conjunto de arestas induzidas pelos valores de p para os vértices em V_p:

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}.$$

Provaremos que os valores de p produzidos pelos algoritmos neste capítulo têm a seguinte propriedade: no término, G_p é uma "árvore de caminhos mínimos" — informalmente, uma árvore enraizada que contém um caminho mínimo da fonte s a todo vértice que pode ser alcançado de s. Uma árvore de caminhos mínimos é semelhante à árvore de busca em largura da Seção 22.2, mas contém caminhos mínimos que partem da fonte definidos em termos de pesos de arestas, em vez de números de arestas. Para sermos precisos, seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$, e considere que G não suponha nenhum ciclo de peso negativo que possa ser alcançado do vértice de fonte $s \in V$, de modo que esses caminhos mínimos são bem definidos. Uma ***árvore de caminhos mínimos*** com raiz em s é um subgrafo dirigido $G' = (V', E')$, onde $V' \subseteq V$ e $E' \subseteq E$, tal que

1. V' é o conjunto de vértices que podem ser alcançados de s em G,
2. G' forma uma árvore enraizada com raiz s e
3. para todo $v \in V'$, o único caminho simples de s a v em G' é um caminho mínimo de s a v em G.

Caminhos mínimos não são necessariamente únicos nem são necessariamente únicas as árvores de caminhos mínimos. Por exemplo, a Figura 24.2 mostra um grafo dirigido ponderado e duas árvores de caminhos mínimos com a mesma raiz.

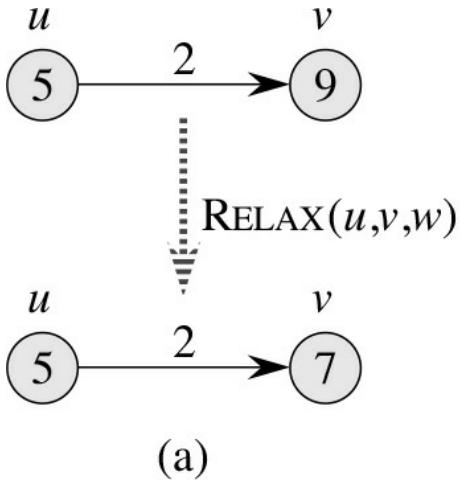
Figura 24.2 (**a**) Um grafo dirigido ponderado com caminhos de peso mínimo que partem da fonte s. (**b**) As arestas sombreadas formam uma árvore de caminhos mínimos com raiz na fonte s. (**c**) Uma outra árvore de caminhos mínimos com a mesma raiz.

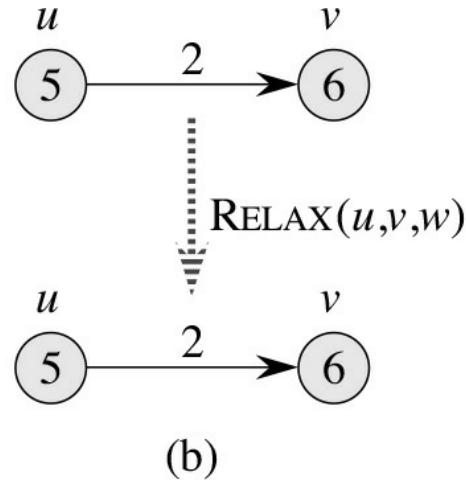
Relaxamento

Os algoritmos neste capítulo usam a técnica de ***relaxamento***. Para cada vértice $v \in V$, mantemos um atributo $v.d$, que é um limite superior para o peso de um caminho mínimo da fonte s a v. Denominamos $v.d$ uma ***estimativa de caminho mínimo***. Inicializamos as estimativas de caminhos mínimos e predecessores pelo seguinte procedimento de tempo $Q(V)|$:

INITIALIZE-SINGLE-SOURCE(G, s)

1 **for** cada vértice $v \in V[G]$
2 $\quad v.d = \infty$
3 $\quad v.\pi = \text{NIL}$
4 $s.d = 0$

Após a inicialização, temos $v.p = \text{NIL}$ para todo $v \in V$, $s.d = 0$ e $v.d = \infty$ para $v \in V - \{s\}$.

O processo de ***relaxar*** uma aresta (u, v) consiste em testar se podemos melhorar o caminho mínimo até v que encontramos até agora passando por u e, em caso positivo, atualizar $v.d$ e $v.p$. Uma etapa de relaxamento[1] pode diminuir o valor da estimativa do caminho mínimo $v.d$ e atualizar o atributo predecessor de v, $v.p$. O seguinte código executa uma etapa de relaxamento para a aresta (u, v) no tempo $O(1)$.

RELAX(u, v, w)

1 **if** $v.d > u.d + w(u, v)$
2 $\quad v.d = u.d + w(u, v)$
3 $\quad v.\pi = u$

A Figura 24.3 mostra dois exemplos de relaxamento de uma aresta: em um deles a estimativa de caminhos mínimos diminui e, no outro, nenhuma estimativa muda.

Cada algoritmo neste capítulo chama INITIALIZE-SINGLE-SOURCE e depois relaxa arestas repetidamente. Além disso, o relaxamento é o único meio de mudar estimativas de caminhos mínimos e predecessores. As diferenças entre os algoritmos apresentados neste capítulo são a quantidade de vezes que relaxam cada aresta e a ordem em que relaxam arestas. O algoritmo de Dijkstra e o algoritmo de caminhos mínimos para grafos acíclicos dirigidos relaxam cada aresta exatamente uma vez. O algoritmo de BellmanFord relaxa cada aresta $|V| - 1$ vezes.

Figura 24.3 Relaxamento de uma aresta (u, v) com peso $w(u, v) = 2$. A estimativa de caminho mínimo de cada vértice aparece dentro do vértice. (**a**) Como $v.d > u.d + w(u, v)$ antes do relaxamento, o valor de $v.d$ diminui. (**b**) Aqui, $v.d \leq u.d + w(u, v)$ antes do relaxamento da aresta e, assim, a etapa de relaxamento deixa $v.d$ como estava.

Propriedades de caminhos mínimos e relaxamento

Para provar a correção dos algoritmos deste capítulo, recorreremos a várias propriedades de caminhos mínimos e relaxamento. Enunciamos essas propriedades aqui, e a Seção 24.5 apresentará a prova formal. Para sua referência, cada propriedade enunciada aqui inclui o número adequado do lema ou corolário da Seção 24.5. As cinco últimas dessas propriedades, que se referem a estimativas de caminhos mínimos ou o subgrafo dos predecessores, supõem implicitamente que o grafo é inicializado com uma chamada a INITIALIZE-SINGLE-SOURCE(G, s) e que o único modo de mudar estimativas de caminhos mínimos e o subgrafo dos predecessores é empregar alguma sequência de etapas de relaxamento.

Desigualdade triangular (Lema 24.10)

Para qualquer aresta $(u, v) \in E$, temos $d(s, v) \leq d(s, u) + w(u, v)$.

Propriedade do limite superior (Lema 24.11)

Sempre temos $v.d \geq d(s, v)$ para todos os vértices $v \in V$ e, tão logo $d[v]$ alcança o valor $d(s, v)$, nunca mais muda.

Propriedade de inexistência de caminho (Corolário 24.12)

Se não existe nenhum caminho de s a v, então sempre temos $v.d = d(s, v) = \infty$.

Propriedade de convergência (Lema 24.14)

Se $s\ u \rightarrow v$ é um caminho mínimo em G para algum $u, v \in V$ e se $u.d = d(s, u)$ em qualquer instante antes de relaxar a aresta (u, v), então $v.d = d(s, v)$ em todos os instantes posteriores.

Propriedade de relaxamento de caminho (Lema 24.15)

Se $p = \langle v_0, v_1, ..., v_k\rangle$ é um caminho mínimo de $s = v_0$ a v_k e relaxamos as arestas de p na ordem (v_0, v_1), (v_1, v_2), ..., $(v_k\text{-}1, v_k)$, então $v_k.d = d(s, v_k)$. Essa propriedade é válida independentemente de quaisquer outras etapas de relaxamento que ocorram, ainda que elas estejam misturadas com relaxamentos das arestas de p.

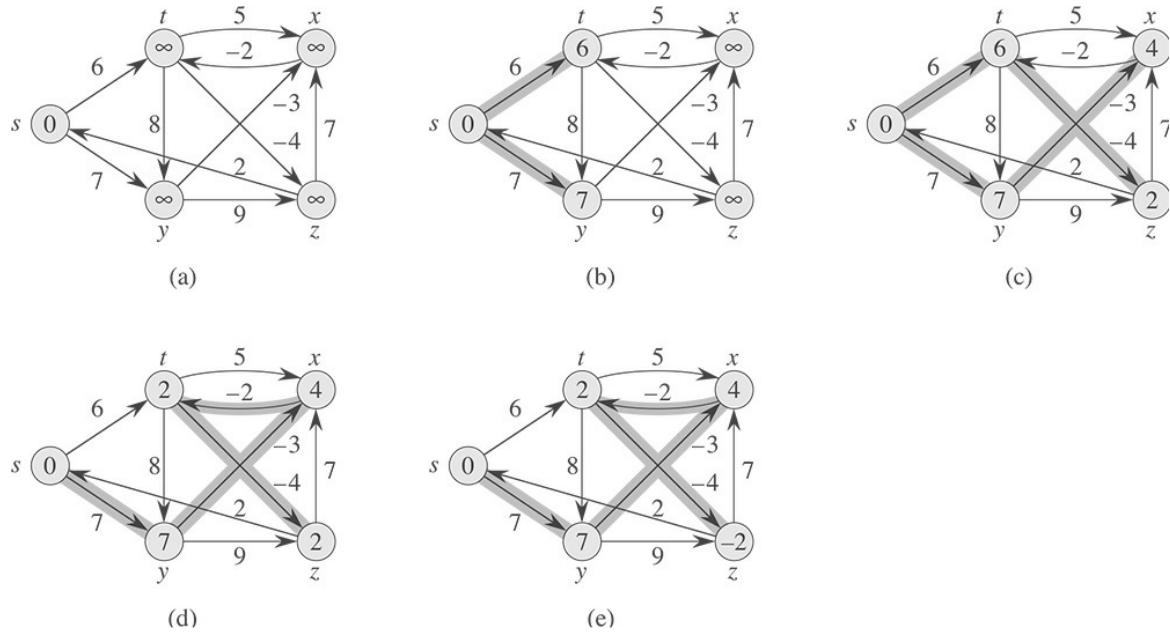
Propriedade do subgrafo dos predecessores (Lema 24.17)

Assim que $v.d = d(s, v)$ para todo $v \in V$, o subgrafo dos predecessores é uma árvore de caminhos mínimos com raiz em s.

Esboço do capítulo

A Seção 24.1 apresenta o algoritmo de Bellman–Ford, que resolve o problema de caminhos mínimos de fonte única no caso geral em que as arestas podem ter peso negativo. O algoritmo de Bellman–Ford é notável por sua simplicidade e tem a vantagem adicional de detectar se um ciclo de peso negativo pode ser alcançado da fonte. A Seção 24.2 dá um algoritmo de tempo linear para calcular caminhos mínimos que partem de uma fonte única em um grafo acíclico dirigido. A Seção 24.3 discute o algoritmo de Dijkstra, cujo tempo de execução é menor que o do algoritmo de Bellman–Ford, mas requer que os pesos de arestas sejam não negativos. A Seção 24.4 mostra como podemos usar o algoritmo de Bellman–Ford para resolver um caso especial de programação linear. Finalmente, a Seção 24.5 prova as propriedades de caminhos mínimos e relaxamento enunciadas anteriormente.

Precisamos de algumas convenções para efetuar cálculos aritméticos com valores infinitos. Adotaremos, para qualquer número real $a \neq -\infty$, $a + \infty = \infty + a = \infty$. Além disso, para tornar nossas demonstrações (ou provas) válidas na presença de ciclos de peso negativo, adotamos, para qualquer número real $a \neq \infty$, $a + (-\infty) = (-\infty) + a = -\infty$.

Todos os algoritmos neste capítulo supõem que o grafo dirigido G é armazenado pela representação de lista de adjacências. Além disso, com cada aresta está armazenado seu peso, de modo que, à medida que percorremos cada lista de adjacências, podemos determinar os pesos de arestas no tempo $O(1)$ por aresta.

24.1 O ALGORITMO DE BELLMAN-FORD

O ***algoritmo de Bellman–Ford*** resolve o problema de caminhos mínimos de fonte única no caso geral no qual os pesos das arestas podem ser negativos. Dado um grafo dirigido ponderado $G = (V, E)$ com fonte s e função peso $w : E \rightarrow$, o algoritmo de Bellman–Ford devolve um valor booleano que indica se existe ou não um ciclo de peso negativo que pode ser alcançado da fonte. Se tal ciclo existe, o algoritmo indica que não há nenhuma solução. Se tal ciclo não existe, o algoritmo produz os caminhos mínimos e seus pesos.

O algoritmo relaxa arestas diminuindo progressivamente uma estimativa $v.d$ do peso de um caminho mínimo da fonte s a cada vértice $v \in V$ até chegar ao peso propriamente dito do caminho mínimo $d(s, v)$. O algoritmo retorna TRUE se e somente se o grafo não contém nenhum ciclo de peso negativo que possa ser alcançado da fonte.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |V[G]| − 1
3     for cada aresta (u, v) ∈ E[G]
4        RELAX(u, v, w)
5  for cada aresta (u, v) ∈ E[G]
6     if v.d > u.d + w(u, v)
7        return FALSE
8  return TRUE
```

A Figura 24.4 mostra a execução do algoritmo de BellmanFord para um grafo com cinco vértices. Depois de inicializar os valores de d e p de todos os vértices na linha 1, o algoritmo faz $|V| - 1$ passagens pelas arestas do grafo. Cada passagem é uma iteração do laço **for** das linhas 2–4 e consiste em relaxar cada aresta do grafo uma vez. As Figuras 24.4(b)–(e) mostram o estado do algoritmo após cada uma das quatro passagens pelas arestas. Depois de executar $|V| - 1$ passagens, as linhas 5–8 verificam se há um ciclo de peso negativo e devolvem o valor booleano adequado. (Um pouco mais adiante veremos por que essa verificação funciona.)

Figura 24.4 Execução do algoritmo de Bellman-Ford. A fonte é o vértice s. Os valores de d aparecem dentro dos vértices, e arestas sombreadas indicam os valores de predecessores; se a aresta (u, v) estiver sombreada, então $v.p = u$. Nesse exemplo específico, cada passagem relaxa as arestas na ordem (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y). (**a**) Situação imediatamente antes da primeira passagem pelas arestas. (**b**)-(**e**) Situação após cada passagem sucessiva pelas arestas. Os valores de d e p na parte (e) são os valores finais. O algoritmo de Bellman-Ford retorna TRUE nesse exemplo.

O algoritmo de BellmanFord é executado no tempo $O(V\ E)$, já que a inicialização na linha 1 demora o tempo $Q(V)$, cada uma das $|V| - 1$ passagens pelas arestas nas linhas 2–4 demora o tempo $Q(E)$, e o laço **for** das linhas 5–7 demora o tempo $O(E)$.

Para provar a correção do algoritmo de BellmanFord, começamos mostrando que, se não existir nenhum ciclo de peso negativo, o algoritmo calcula pesos de caminhos mínimos corretos para todos os vértices que podem ser alcançados da fonte.

Lema 24.2

Seja $G = (V, E)$ um grafo dirigido ponderado com fonte s e função peso $w : E \rightarrow$, e suponha que G não contenha nenhum ciclo de peso negativo que possa ser alcançado de s. Então, após as $|V| - 1$ iterações do laço **for** das linhas 2–4 de BELLMAN-FORD, temos $v.d = \text{d}(s, v)$ para todos os vértices v que podem ser alcançados de s.

Prova Provamos o lema apelando para a propriedade de relaxamento de caminho. Considere qualquer vértice v que possa ser alcançado de s e seja $p = \langle v_0, v_1, ..., v_k\rangle$, onde $v_0 = s$ e $v_k = v$, qualquer caminho mínimo de s a v. Como os caminhos mínimos são simples, p tem no máximo $|V| - 1$ arestas e, assim, $k \leq |V| - 1$. Cada uma das $|V| - 1$ iterações do laço **for** das linhas 2–4 relaxa todas as $|E|$ arestas. Entre as arestas relaxadas na i–ésima iteração, para $i = 1, 2, ..., k$, está $(v_{i} - {}^{1}, v_{i})$. Então, pela propriedade de relaxamento de caminho, $v.d = v_k.d = d(s, v_k) = d(s, v)$.

Corolário 24.3

Seja $G = (V, E)$ um grafo dirigido ponderado com vértice fonte s e função peso $w : E \rightarrow$ e suponha que G não contenha nenhum ciclo negativo que possa ser alcançado de s. Então, para cada vértice $v \in V$, existe um caminho de s a v se e somente se BELLMAN-FORD termina com $v.d < \infty$ quando é executado em G.

Prova A prova fica para o Exercício 24.1–2

Teorema 24.4 (Correção do algoritmo de Bellman-Ford)

Considere o algoritmo de BELLMAN-FORD executado para um grafo dirigido ponderado $G = (V, E)$ com fonte s e função peso $w : E \rightarrow$. Se G não contém nenhum ciclo de peso negativo que possa ser alcançado de s, então o algoritmo retorna TRUE, temos $v.d = d(s, v)$ para todos os vértices $v \in V$, e o subgrafo predecessor G_p é uma árvore de caminhos mínimos com raiz em s. Se G contém um ciclo de peso negativo que possa ser alcançado de s, então o algoritmo retorna FALSE.

Prova Suponha que o grafo G não contenha nenhum ciclo de peso negativo que possa ser alcançado da fonte s. Primeiro, provamos a afirmação de que, no término, $v.d = d(s, v)$ para todos os vértices $v \in V$. Se o vértice v pode ser alcançado de s, então o Lema 24.2 prova essa afirmação. Se v não pode ser alcançado de s, a afirmação decorre da propriedade da inexistência de caminho. Portanto, a afirmação está provada. A propriedade do subgrafo dos predecessores, juntamente com a afirmação, implica que G_p é uma árvore de caminhos mínimos. Agora, usamos a afirmação para mostrar que BELLMAN-FORD retorna TRUE. No término, temos para todas as arestas $(u, v) \in E$,

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \qquad \text{(pela desigualdade triangular)} \\ &= u.d + w(u, v) \ , \end{aligned}$$

e, assim, nenhum dos testes na linha 6 faz BELLMAN-FORD retornar FALSE. Então, ele retorna TRUE.

Agora, suponha que o grafo G contenha um ciclo de peso negativo que possa ser alcançado da fonte s; seja esse ciclo $c = \langle v_0, v_1, ..., v_k \rangle$, onde $v_0 = v_k$. Então,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \, . \qquad (24.1)$$

Considere, por contradição, que o algoritmo de BellmanFord retorne TRUE. Assim, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ para $i = 1, 2, ..., k$. Somando as desigualdades em torno do ciclo c temos

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i). \end{aligned}$$

Como $v_0 = v_k$, cada vértice em c aparece exatamente uma vez em cada um dos somatórios e $\sum_{i=1}^{k} v_i.d$, e $\sum_{i=1}^{k} v_{i-1}.d$, portanto

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$$

Além disso, pelo Corolário 24.3, $v_i.d$ é finito para $i = 1, 2, ..., k$. Assim,

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) \, ,$$

o que contradiz a desigualdade (24.1). Concluímos que o algoritmo de BellmanFord retorna TRUE se o grafo G não contém nenhum ciclo de peso negativo que possa ser alcançado da fonte, e FALSE em caso contrário.

Exercícios

24.1-1 Execute o algoritmo de BellmanFord no grafo dirigido da Figura 24.4, usando o vértice z como fonte. Em cada passagem, relaxe arestas na mesma ordem da figura e mostre os valores de d e p após cada passagem. Agora, mude o peso da aresta (z, x) para 4 e execute o algoritmo novamente, usando s como fonte.

24.1-2 Prove o Corolário 24.3.

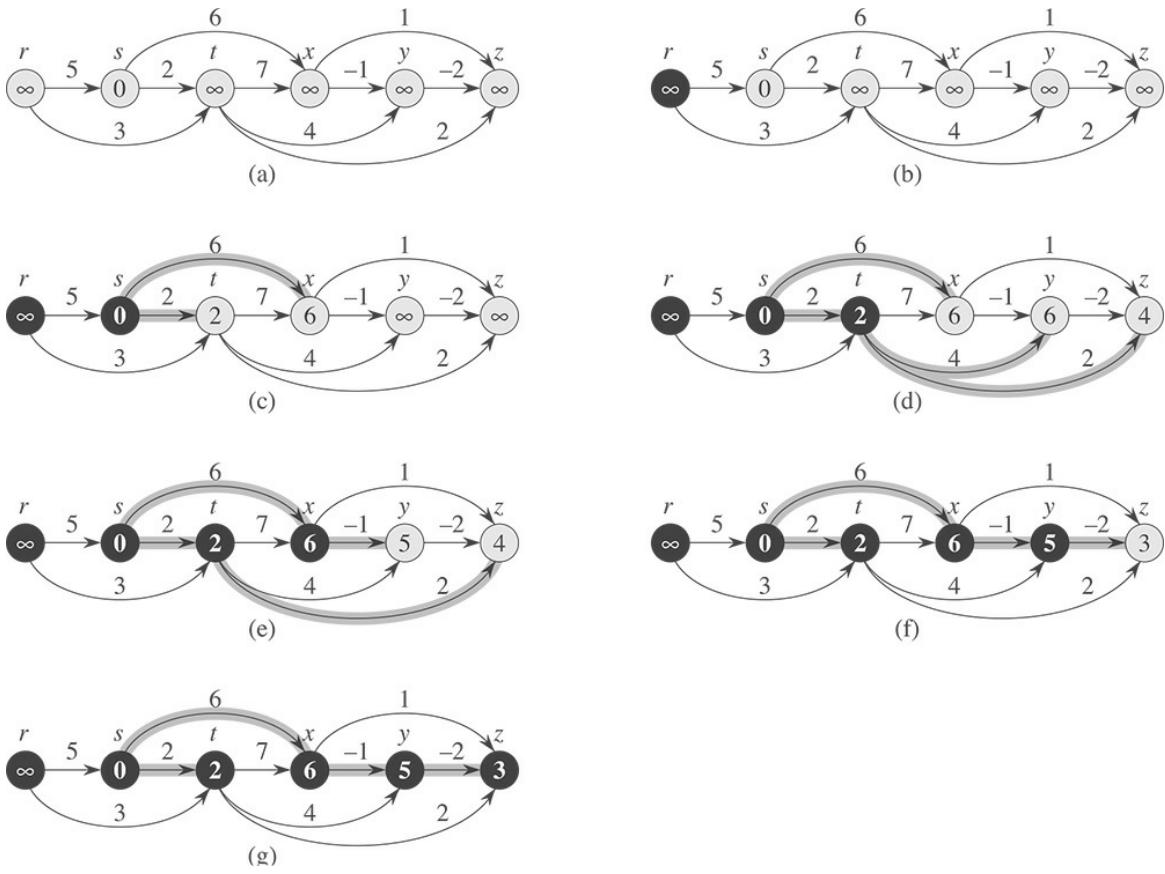
24.1-3 Dado um grafo dirigido ponderado $G = (V, E)$ sem ciclos de peso negativo, seja m o máximo para todos os vértices $u, v \in V$ do número mínimo de arestas em um caminho mínimo da fonte s até v. (Aqui, o caminho mínimo é por peso, não pelo número de arestas.) Sugira uma mudança simples no algoritmo de BellmanFord que permita terminá–lo em $m + 1$ passagens, ainda que m não seja conhecido com antecedência.

24.1-4 Modifique o algoritmo de BellmanFord de modo que ele termine com $v.d$ como $-\infty$ para todos os vértices v para os quais existe um ciclo de peso negativo em algum caminho da fonte até v.

24.1-5 ★ Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$. Dê um algoritmo de tempo $O(V E)$ para encontrar, para cada vértice $v \in V$, o valor $d^*(v) = \min_{u \in V} \{d(u, v)\}$.

24.1-6 ★ Suponha que um grafo dirigido ponderado $G = (V, E)$ tenha um ciclo de peso negativo. Dê um algoritmo eficiente para produzir uma lista de vértices de tal ciclo. Prove que seu algoritmo é correto.

24.2 CAMINHOS MÍNIMOS DE FONTE ÚNICA EM GRAFOS ACÍCLICOS DIRIGIDOS

Relaxando as arestas de um gad (grafo acíclico dirigido) ponderado $G = (V, E)$, de acordo com uma ordenação topológica de seus vértices, podemos calcular caminhos mínimos de uma fonte única no tempo $Q(V + E)$. Caminhos mínimos são sempre bem definidos em um gad já que, mesmo que existam arestas de peso negativo, não deve existir nenhum ciclo de peso negativo.

O algoritmo começa ordenando topologicamente o gad (veja a Seção 22.4) para impor uma ordenação linear para os vértices. Se o gad contém um caminho do vértice u ao vértice v, então u precede v na ordem topológica. Fazemos apenas uma passagem pelos vértices na sequência ordenada topologicamente À medida que processamos cada vértice, relaxamos cada aresta que sai do vértice.

```
DAG-SHORTEST-PATHS (G, w, s)

1   ordenar topologicamente os vértices de G
2   INITIALIZE-SINGLE-SOURCE (G, s)
3   for cada vértice u tomado em ordem topológica
4      for cada vértice v ∈ Adj[u]
5          RELAX(u, v, w)
```

A Figura 24.5 mostra a execução desse algoritmo.

É fácil analisar o tempo de execução desse algoritmo. Como mostra a Seção 22.4, a ordenação topológica da linha 1 demora o tempo $Q(V + E)$. A chamada de INITIALIZE-SINGLE-SOURCE na linha 2 demora o tempo $Q(V)$. O laço **for** das linhas 3–5 executa uma iteração por vértice. No total, o laço **for** das linhas 4–5 relaxa cada vértice exatamente uma vez. (Usamos aqui uma análise agregada.) Como cada iteração do laço **for** demora o tempo $Q(1)$, o tempo total de execução é $Q(V + E)$, que é linear em relação ao tamanho de uma representação de lista de adjacências do grafo.

Figura 24.5 Execução do algoritmo para caminhos mínimos em um grafo acíclico dirigido. Os vértices são ordenados topologicamente da esquerda para a direita. O vértice da fonte é s. Os valores de d aparecem dentro dos vértices, e arestas sombreadas indicam os valores de p. (**a**) Situação antes da primeira iteração do laço **for** das linhas 3-5. (**b**)-(**g**) Situação após cada iteração do laço **for** das linhas 3-5. O vértice que acabou de ser pintado de preto em cada iteração foi usado como u naquela iteração. Os valores mostrados na parte (g) são os valores finais.

O teorema apresentado a seguir mostra que o procedimento DAG-SHORTEST-PATHS calcula corretamente os caminhos mínimos.

Teorema 24.5

Se um grafo dirigido ponderado $G = (V, E)$ tem vértice de fonte s e nenhum ciclo, então no término do procedimento DAG-SHORTEST-PATHS, $v.d = d(s, v)$ para todos os vértices $v \in V$, e o subgrafo dos predecessores G_p é uma árvore de caminhos mínimos.

Prova Primeiro, mostramos que $v.d = d(s, v)$ para todos os vértices $v \in V$ no término. Se v não pode ser alcançado de s, então $v.d = d(s, v) = \infty$ pela propriedade da inexistência de caminho. Agora, suponha que v possa ser alcançado de s, de modo que existe um caminho mínimo $p = \langle v_0, v_1, ..., v_k \rangle$, onde $v_0 = s$ e $v_k = v$. Como processamos os vértices em sequência ordenada topologicamente, relaxamos as arestas em p na ordem (v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k). A propriedade de relaxamento de caminho implica que $d[v_i] = d(s, v_i)$ no término para $i = 0, 1, ..., k$. Finalmente, pela propriedade de subgrafo predecessor, G_p é uma árvore de caminhos mínimos.

Uma aplicação interessante desse algoritmo surge na determinação de caminhos críticos na análise de ***diagramas PERT***.[2] As arestas representam serviços que devem ser executados, e os pesos de arestas representam os tempos necessários para a execução de determinados serviços. Se a aresta (u, v) entra no vértice v e a aresta (v, x) sai de v, então o serviço (u, v) deve ser executado antes do serviço (v, x). Um caminho nesse gad representa uma sequência de

serviços que devem ser executados em determinada ordem. Um ***caminho crítico*** é um caminho *de comprimento máximo* no gad, correspondente ao tempo máximo para a execução de qualquer sequência de serviços. Assim, o peso de um caminho crítico dá um limite inferior para o tempo total de execução de todos os serviços. Podemos encontrar um caminho crítico de duas maneiras:

- tornando negativos os pesos de arestas e executando DAG-SHORTEST-PATHS ou
- executando DAG-SHORTEST-PATHS, substituindo "∞" por "-∞" na linha 2 de INITIALIZE-SINGLE-SOURCE e ">" por "<" no procedimento RELAX.

Exercícios

24.2-1 Execute DAG-SHORTEST-PATHS para o grafo dirigido da Figura 24.5, usando o vértice r como fonte.

24.2-2 Suponha que mudamos a linha 3 de DAG-SHORTEST-PATHS para 3 **for** os primeiros $|V| - 1$ vértices, tomados em ordem topológica

Mostre que o procedimento continuaria correto.

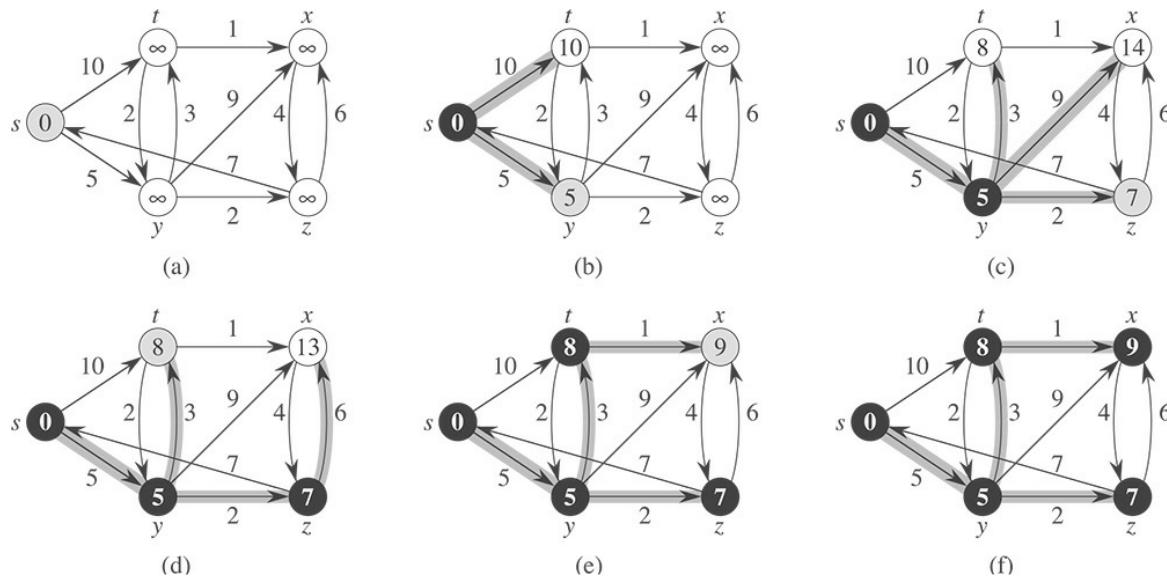
24.2-3 A formulação de diagramas PERT que citamos não é muito natural. Mais natural seria uma estrutura na qual os vértices representassem serviços e as arestas representassem restrições de sequenciamento; isto é, a aresta (u, v) indicaria que o serviço u deve ser executado antes do serviço v. Então, atribuiríamos pesos a vértices, e não a arestas. Modifique o procedimento DAG-SHORTEST-PATHS de modo que ele encontre um caminho de comprimento máximo em um grafo acíclico dirigido com vértices ponderados em tempo linear.

24.2-4 Dê um algoritmo eficiente para contar o número total de caminhos em um grafo acíclico dirigido. Analise seu algoritmo.

24.3 ALGORITMO DE DIJKSTRA

O algoritmo de Dijkstra resolve o problema de caminhos mínimos de fonte única em um grafo dirigido ponderado $G = (V, E)$ para o caso no qual todos os pesos de arestas são não negativos. Então, nesta seção suporemos que $w(u, v) \geq 0$ para cada aresta $(u, v) \in E$. Como veremos, com uma boa implementação, o tempo de execução do algoritmo de Dijkstra é inferior ao do algoritmo de BellmanFord.

O algoritmo de Dijkstra mantém um conjunto S de vértices cujos pesos finais de caminhos mínimos que partem da fonte s já foram determinados. O algoritmo seleciona repetidamente o vértice $u \in V - S$ que tem a mínima estimativa do caminho mínimo, adiciona u a S e relaxa todas as arestas que saem de u. Na implementação a seguir, usamos uma fila de prioridades mínimas Q de vértices cujas chaves são os valores de d.

DIJKSTRA(G, w, s)

1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S = \emptyset$
3 $Q = V[G]$
4 **while** $Q \neq \emptyset$
5   $u =$ EXTRACT-MIN(Q)
6   $S = S \cup \{u\}$
7   **for cada vértice** $v \in G.Adj[u]$
8   RELAX(u, v, w)

O algoritmo de Dijkstra relaxa arestas, como mostra a Figura 24.6. A linha 1 inicializa os valores de *d* e *p* do modo usual, e a linha 2 inicializa o conjunto *S* como o conjunto vazio. O algoritmo mantém o invariante $Q = V - S$ no início de cada iteração do laço **while** das linhas 4–8. A linha 3 inicializa a fila de prioridades mínimas *Q* para conter todos os vértices em *V*; visto que *S* = /0 nesse momento, o invariante é verdadeiro após a linha 3. Em cada passagem pelo laço **while** das linhas 48, a linha 5 extrai um vértice *u* de $Q = V - S$ e a linha 6 o adiciona ao conjunto *S*, mantendo assim o invariante. (Na primeira passagem por esse laço, $u = s$.) Portanto, o vértice *u* tem a menor estimativa de caminhos mínimos em comparação com qualquer vértice em $V - S$. Então, as linhas 78 relaxam cada aresta (u, v) que sai de *u*, atualizando assim a estimativa *v.d* e o predecessor *v.p* se, passando por *u*, pudermos melhorar o caminho mínimo até *v* que encontramos até aqui. Observe que o algoritmo nunca insere vértices em *Q* após a linha 3 e que cada vértice é extraído de *Q* e adicionado a *S* exatamente uma vez, de modo que o laço **while** das linhas 4–8 itera exatamente $|V|$ vezes.

Como o algoritmo de Dijkstra sempre escolhe o vértice "mais leve" ou "mais próximo" em $V - S$ para adicionar ao conjunto *S*, podemos dizer que ele utiliza uma estratégia gulosa. O Capítulo 16 dá uma explicação detalhada de estratégias gulosas, mas você não precisa ler aquele capítulo para entender o algoritmo de Dijkstra. Estratégias gulosas nem sempre produzem resultados ótimos em geral, mas, como mostram o teorema a seguir e seu corolário, o algoritmo de Dijkstra realmente calcula caminhos mínimos. A chave é mostrar que, cada vez que o algoritmo insere um vértice *u* no conjunto *S*, temos $u.d = d(s, u)$.

Figura 24.6 Execução do algoritmo de Dijkstra. A fonte s é o vértice mais à esquerda. As estimativas de caminho mínimo aparecem dentro dos vértices, e as arestas sombreadas indicam valores de predecessores. Vértices pretos estão no conjunto S e vértices brancos estão na fila de prioridades mínimas $Q = V - S$. (**a**) Situação imediatamente antes da primeira iteração do laço **while** das linhas 4-8. O vértice sombreado tem o valor d mínimo e é escolhido como vértice u na linha 5. (**b**)-(**f**) Situação após cada iteração sucessiva do laço **while**. O vértice sombreado em cada parte é escolhido como vértice u na linha 5 da próxima iteração. Os valores de d e p mostrados na parte (f) são os valores finais.

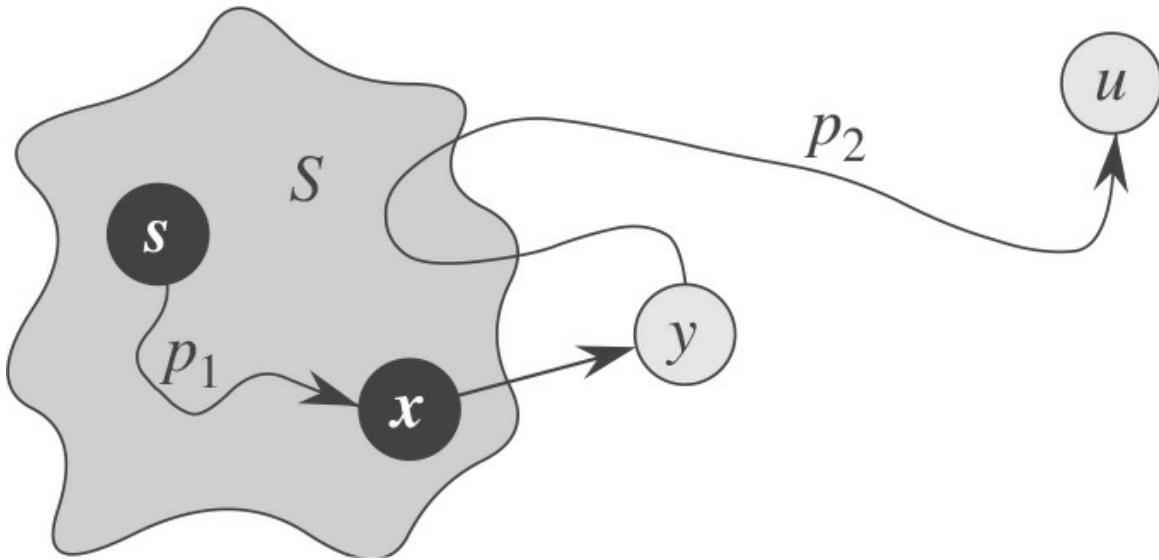
Teorema 24.6 (Correção do algoritmo de Dijkstra)

O algoritmo de Dijkstra, se executado para um grafo dirigido ponderado $G = (V, E)$ com função peso não negativa w e fonte s, termina com $u.d = d(s, u)$ para todos os vértices $u \in V$.

Prova Usamos o seguinte invariante de laço:

No início de cada iteração do laço **while** das linhas 4–8, $v.d = d(s, v)$ para cada vértice $v \in S$.

Basta mostrar que, para cada vértice $u \in V$, temos $u.d = d(s, u)$ no momento em que u é adicionado ao conjunto S. Uma vez mostrado que $u.d = d(s, u)$, recorremos à propriedade do limite superior para mostrar que a igualdade é válida em todos os momentos daí em diante.

Inicialização: Inicialmente, $S = \varnothing$ e, assim, o invariante é trivialmente verdadeiro.

Manutenção: Desejamos mostrar que, em cada iteração, $u.d = d(s, u)$ para o vértice adicionado ao conjunto S. Por contradição, seja u o primeiro vértice para o qual $u.d \neq d(s, u)$ quando ele é adicionado ao conjunto S. Concentraremos nossa atenção na situação existente no início da iteração do laço **while** quando u é adicionado a S e deduzimos a contradição $u.d = d(s, u)$ naquele instante examinando um caminho mínimo de s a u. Devemos ter $u \neq s$ porque s é o primeiro vértice adicionado ao conjunto S e $s.d = d(s, s) = 0$ naquele instante. Como $u \neq s$, temos também que $S \neq /0$ exatamente antes de u ser adicionado a S. Deve existir algum caminho de s a u, senão $u.d = d(s, u) = \infty$ pela propriedade da inexistência de caminho, o que violaria nossa hipótese $u.d \neq d(s, u)$. Como há pelo menos um caminho, existe um caminho mínimo p de s a u. Antes de se adicionar u a S, o caminho p conecta um vértice em S, isto é, s a um vértice em $V - S$, isto é, u. Vamos considerar o primeiro vértice y ao longo de p tal que $y \in V - S$, e seja $x \in S$ o predecessor de y. Assim, como mostra a Figura 24.7, podemos decompor o caminho p em $s\ p_1\ x \rightarrow y\ p_2\ u$. (Qualquer dos caminhos p_1 ou p_2 pode não ter nenhuma aresta.) Afirmamos que $y.d = d(s, y)$ quando u é adicionado a S. Para provar essa afirmação, observe que $x \in S$. Então, como u foi escolhido como o primeiro vértice para o qual $u.d \neq d(s, u)$ quando foi adicionado a S, tínhamos $x.d = d(s, x)$ quando x foi adicionado a S. A aresta (x, y) foi relaxada naquele momento, e a afirmação decorre da propriedade de convergência. Agora, podemos obter uma contradição para provar que $d[u] = d(s, u)$. Como y aparece antes de u em um caminho mínimo de s a u e todos os pesos de arestas são não negativos (especialmente os das arestas do caminho p_2), temos $d(\text{s}, y) \leq d(s, u)$ e, assim,

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq u.d \qquad \text{(pela propriedade do limite superior)}\,. \end{aligned} \tag{24.2}$$

Porém, como os vértices u e y estavam em $V - S$ quando u foi escolhido na linha 5, temos $u.d \leq y.d]$. Assim, as duas desigualdades em (24.2) são de fato igualdades, o que dá

Figura 24.7 A prova do Teorema 24.6. O conjunto S é não vazio imediatamente antes de o vértice u ser adicionado a ele. Decompomos um caminho mínimo p da fonte s ao vértice u em $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, onde y é o primeiro vértice no caminho que não está em S e $x \in S$ precede imediatamente y. Os vértices x e y são distintos, mas poderíamos ter $s = x$ ou $y = u$. O caminho p_2 pode reentrar ou não no conjunto S.

$$y.d = \delta(s, y) = \delta(s, u) = u.d \; .$$

Consequentemente, $u.d = d(s, u)$, o que contradiz nossa escolha de u. Concluímos que $u.d = d(s, u)$ quando u é adicionado a S e que essa igualdade é mantida por todo o tempo daí em diante.

Término: No término, $Q = /0$ que, juntamente com nosso invariante anterior, $Q = V - S$, implica $S = V$. Assim, $u.d = d(s, u)$ para todos os vértices $u \in V$.

Corolário 24.7

Se executarmos o algoritmo de Dijkstra para um grafo dirigido ponderado $G = (V, E)$ com função peso não negativa w e fonte s, então, no término, o subgrafo predecessor G_p será uma árvore de caminhos mínimos com raiz em s.

Prova Imediata pelo Teorema 24.6 e pela propriedade do subgrafo dos predecessores.

Análise

Qual é a rapidez do algoritmo de Dijkstra? Ele mantém a fila de prioridades mínimas Q chamando três operações de filas de prioridades: INSERT (implícita na linha 3), EXTRACT-MIN (linha 5) e DE- CREASE-KEY (implícita em RELAX, que é chamada na linha 8). O algoritmo chama INSERT e EXTRACT-MIN uma vez por vértice. Como cada vértice $u \in V$ é adicionado ao conjunto S exatamente uma vez, cada aresta na lista de adjacências $Adj[u]$ é examinada no laço **for** das linhas 7–8 exatamente uma vez durante o curso do algoritmo. Visto que o número total de arestas em todas as listas de adjacências é $|E|$, esse laço **for** itera um total de $|E|$ vezes e, assim, o algoritmo chama DECREASE-KEY no máximo $|E|$ vezes no total. (Observe, mais uma vez, que estamos usando análise agregada.)

O tempo de execução do algoritmo de Dijkstra depende de como implementamos a fila de prioridades mínimas. Considere primeiro o caso no qual mantemos a fila de prioridades mínimas aproveitando que os vértices são numerados de 1 a $|V|$. Simplesmente armazenamos $v.d$ na $v$ésima entrada de um arranjo. Cada operação INSERT e DECREASE-KEY

demora o tempo $O(1)$, e cada operação Extract-Min demora o tempo $O(V)$ (já que temos de executar busca no arranjo inteiro), dando um tempo total $O(V_2 + E) = O(V_2)$.

Se o grafo é suficientemente esparso — em particular, $E = o(V_2/\lg V)$ — podemos melhorar o algoritmo implementando a fila de prioridades mínimas com um heap de mínimo binário. (Como discutimos na Seção 6.5, a implementação deve garantir que os vértices e os elementos do heap correspondentes mantêm apontadores um para o outro.) Então, cada operação Extract-Min demora o tempo $O(\lg V)$. Como antes, há $|V|$ dessas operações. O tempo para construir o heap de mínimo binário é $O(V)$. Cada operação Decrease-Key demora o tempo $O(\lg V)$ e ainda há, no máximo, $|E|$ dessas operações. Portanto, o tempo de execução total é $O((V + E) \lg V)$, que é $O(E \lg V)$ se todos os vértices podem ser alcançados da fonte. Esse tempo de execução é uma melhoria em relação ao tempo $O(V_2)$ de implementação direta se $E = o(V_2/\lg V)$.

Na verdade, podemos conseguir um tempo de execução $O(V \lg V + E)$ implementando a fila de prioridades mínimas com um heap de Fibonacci (veja o Capítulo 19). O custo amortizado de cada uma das $|V|$ operações Extract-Min é $O(\lg V)$, e cada chamada Decrease-Key, cujo número máximo é $|E|$, demora apenas o tempo amortizado $O(1)$. Historicamente, o desenvolvimento de heaps de Fibonacci foi motivado pela observação de que o algoritmo de Dijkstra, normalmente faz muito mais chamadas Decrease-Key que chamadas Extract-Min; portanto, qualquer método de redução do tempo amortizado de cada operação Decrease-Key para $o(\lg V)$ sem aumentar o tempo amortizado de Extract-Min produzirá uma implementação assintoticamente mais rápida do que a que utilize heaps binários.

O algoritmo de Dijkstra é parecido com o algoritmo de busca em largura (veja a Seção 22.2) e também com o algoritmo de Prim para calcular árvores geradoras mínimas (veja a Seção 23.2). É semelhante à busca em largura no sentido de que o conjunto S corresponde ao conjunto de vértices pretos em uma busca em largura; exatamente como os vértices em S têm seus pesos finais de caminhos mínimos, os vértices pretos em uma busca em largura têm suas distâncias em largura corretas. O algoritmo de Dijkstra é semelhante ao algoritmo de Prim no sentido de que ambos usam uma fila de prioridades mínimas para encontrar o vértice "mais leve" fora de um conjunto dado (o conjunto S no algoritmo de Dijkstra e a árvore que está sendo desenvolvida no algoritmo de Prim), adicionam esse vértice ao conjunto e ajustam os pesos dos vértices restantes fora do conjunto de acordo com o resultado dessas operações.

Exercícios

24.3-1 Execute o algoritmo de Dijkstra para o grafo dirigido da Figura 24.2, primeiro usando o vértice s como fonte e depois usando o vértice z como fonte. No estilo da Figura 24.6, mostre os valores de d e p e os vértices no conjunto S após cada iteração do laço **while**.

24.3-2 Dê um exemplo simples de grafo dirigido com arestas de peso negativo para o qual o algoritmo de Dijkstra produz respostas incorretas. Por que a prova do Teorema 24.6 não funciona quando são permitidas arestas de peso negativo?

24.3-3 Suponha que mudemos a linha 4 do algoritmo de Dijkstra para o seguinte:

4 **while** $|Q| > 1$

Essa mudança faz o laço **while** ser executado $|V| - 1$ vezes em lugar de $|V|$ vezes. Esse algoritmo proposto é correto?

24.3-4 O professor Gaedel escreveu um programa que, diz ele, implementa o algoritmo de Dijkstra. O programa produz $v.d$ e $v.p$ para cada vértice $v \in V$. Dê um algoritmo de tempo $O(V + E)$ para verificar a saída do programa do professor. O algoritmo deve determinar se os atributos d e p são compatíveis com os de alguma árvore de caminhos mínimos. Suponha que todos os pesos de arestas são não negativos.

24.3-5 O professor Newman acha que criou uma prova simples da correção do algoritmo para o algoritmo de Dijkstra. Diz ele que o algoritmo de Dijkstra relaxa as arestas de todos os caminhos mínimos no grafo na ordem em que eles aparecem no caminho e que, portanto, a propriedade de relaxamento de caminho se aplica a todos os vértices que podem ser alcançados da fonte. Mostre que o professor está enganado ao construir um grafo dirigido para o qual o algoritmo de Dijkstra poderia relaxar as arestas de uma caminho mínimo fora da ordem.

24.3-6 Temos um grafo dirigido $G = (V, E)$ no qual cada aresta $(u, v) \in E$ tem um valor associado $r(u, v)$, que é um número real na faixa $0 \leq r(u, v) \leq 1$ que representa a confiabilidade de um canal de comunicação do vértice u ao vértice v. Interpretamos $r(u, v)$ como a probabilidade de o canal de u a v não falhar e consideramos que essas probabilidades são independentes. Dê um algoritmo eficiente para encontrar o caminho mais confiável entre dois vértices dados.

24.3-7 Seja $G = (V, E)$ um grafo dirigido ponderado com função peso positivo $w : E \rightarrow \{1, 2, ..., W\}$ para algum inteiro positivo W, e suponha que não haja dois vértices que tenham os mesmos pesos de caminhos mínimos que partem do vértice de fonte s. Agora, suponha que definimos um grafo dirigido não ponderado $G' = (V \cup V', E')$ substituindo cada aresta $(u, v) \in E$ por $w(u, v)$ arestas de peso unitário em série. Quantos vértices G' tem? Suponha agora que executemos uma busca em largura em G'. Mostre que a ordem em que G' pinta os vértices em V de preto na busca em largura é igual à ordem em que o algoritmo de Dijkstra extrai os vértices de V da fila de prioridades quando executado em G.

24.3-8 ★ Seja $G = (V, E)$ um grafo dirigido ponderado com função peso w não negativo $: E \rightarrow \{0, 1, ..., W\}$ para algum inteiro não negativo W. Modifique o algoritmo de Dijkstra para calcular os caminhos mínimos que partem de determinado vértice de fonte s no tempo $O(W V + E)$.

24.3-9 ★ Modifique seu algoritmo do Exercício 24.38 para ser executado no tempo $O((V + E) \lg W)$. (*Sugestão*: Quantas estimativas distintas de caminhos mínimos podem existir em $V - S$ em qualquer instante?)

24.3-10 Suponha que tenhamos um grafo dirigido ponderado $G = (V, E)$ no qual as arestas que saem do vértice fonte s podem ter pesos negativos, todos os outros pesos de arestas são não negativos e não existe nenhum ciclo de peso negativo. Demonstre que o algoritmo de Dijkstra encontra corretamente caminhos mínimos que partem de s nesse grafo.

24.4 Restrições de diferença e caminhos mínimos

O Capítulo 29 estuda o problema geral de programação linear no qual desejamos otimizar uma função linear sujeita a um conjunto de desigualdades lineares. Nesta seção, investigaremos um caso especial de programação linear que reduzimos a encontrar caminhos mínimos que partem de uma única fonte. Então, podemos resolver o problema de caminhos mínimos de fonte única resultante executando o algoritmo de BellmanFord e, por consequência, resolvemos também o problema de programação linear.

Programação linear

No ***problema de programação linear*** geral, temos uma matriz $m \times n$ A, um vetor b de m elementos, e um vetor c de n elementos Desejamos encontrar um vetor x de n elementos que maximize a ***função objetivo*** $\sum_{i=1}^{n} c_i x_i$ sujeita às m restrições dadas por $Ax \leq b$.

Embora o algoritmo simplex focalizado no Capítulo 29 nem sempre seja executado em tempo polinomial em relação ao tamanho de sua entrada, há outros algoritmos de programação linear que são de tempo polinomial. Apresentamos aqui duas razões para entender a formulação de problemas de programação linear. A primeira é que, se soubermos que podemos expressar determinado problema como um problema de programação linear de tamanho polinomial, então temos imediatamente um algoritmo de tempo polinomial para resolver o problema. A segunda é que existem algoritmos mais rápidos para muitos casos especiais de programação linear. Por exemplo, o problema do caminho mínimo para um par (Exercício 24.44) e o problema de fluxo máximo (Exercício 26.15) são casos especiais de programação linear.

Na realidade, às vezes, não nos importamos com a função objetivo; o que queremos é encontrar alguma ***solução viável***, isto é, qualquer vetor x que satisfaça $Ax \leq b$ ou determinar que não existe nenhuma solução viável. Focalizaremos um desses ***problemas de viabilidade***.

Sistemas de restrições de diferença

Em um ***sistema de restrições de diferença***, cada linha da matriz de programação linear A contém um 1 e um –1, e todas as outras entradas de A são 0. Assim, as restrições dadas por $Ax \leq b$ são um conjunto de m ***restrições de diferença*** que envolvem n incógnitas, no qual cada restrição é uma desigualdade linear simples da forma

$$x_j - x_i \leq b_k ,$$

onde $1 \leq i, j \leq n$ e $1 \leq k \leq m$.

Por exemplo, considere o problema de encontrar um vetor de cinco elementos $x = (x_i)$ que satisfaz

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

Esse problema é equivalente a determinar valores para as incógnitas x_1, x_2, x_3, x_4, x_5, que satisfaçam as seguintes oito restrições de diferença:

$$x_1 - x_2 \leq 0 , \quad (24.3)$$
$$x_1 - x_5 \leq -1 , \quad (24.4)$$
$$x_2 - x_5 \leq 1 , \quad (24.5)$$
$$x_3 - x_1 \leq 5 , \quad (24.6)$$
$$x_4 - x_1 \leq 4 , \quad (24.7)$$
$$x_4 - x_3 \leq -1 , \quad (24.8)$$
$$x_5 - x_3 \leq -3 , \quad (24.9)$$
$$x_5 - x_4 \leq -3 . \quad (24.10)$$

Uma solução para esse problema é $x = (–5, –3, 0 ,–1, –4)$, como podemos verificar diretamente examinando cada desigualdade. Na verdade, esse problema tem mais de uma solução. Uma outra é $x' = (0, 2, 5, 4, 1)$. Essas duas soluções estão relacionadas: cada componente de x' é cinco unidades maior que o componente correspondente de x. Esse fato não é mera coincidência.

Lema 24.8

Seja $x = (x_1, x_2, \cdots, x_n)$ uma solução para um sistema $Ax \le b$ de restrições de diferença e seja d qualquer constante. Então, $x + d = (x_1 + d, x_2 + d, ..., x_n + d)$ é também uma solução para $Ax \le b$.

Prova Para cada x_i e x_j, temos $(x_j + d) - (x_i + d) = x_j - x_i$. Assim, se x satisfaz $Ax \le b$, $x + d$ também satisfaz essa desigualdade.

Sistemas de restrições de diferença ocorrem em muitas aplicações diferentes. Por exemplo, as incógnitas x_i podem ser os tempos em que certos eventos devem ocorrer. Cada restrição afirma que deve decorrer no mínimo certa quantidade de tempo ou no máximo certa quantidade de tempo entre dois eventos. Por exemplo, os eventos poderiam ser serviços que devem ser executados durante a montagem de um produto. Se aplicarmos no tempo x_1 um adesivo que demora duas horas para agir e tivermos de esperar essas duas horas para instalar uma peça no tempo x_2, então temos a seguinte restrição: $x_2 \ge x_1 + 2$ ou, o que é equivalente, $x_1 - x_2 \le -2$. Alternativamente, poderíamos determinar que a peça deve ser instalada depois da aplicação do adesivo, porém, em vez de de espererarmos as duas horas previstas até o adesivo agir, devemos esperar somente metade desse tempo. Nesse caso, obtemos o par de restrições x_2 x_1 e $x_2 \le x_1 + 1$ ou, o que é equivalente, $x_1 - x_2 \le 0$ e $x_2 - x_1 \le 1$.

Grafos de restrições

Podemos interpretar sistemas de restrições de diferença adotando um ponto de vista de teoria dos grafos. Em um sistema $Ax \le b$ de restrições de diferença, consideramos a matriz de programação linear A $n \times m$ como a transposta de uma matriz de incidência (veja o Exercício 22.1–7) para um grafo com n vértices e m arestas. Cada vértice v_i no grafo, para $i = 1, 2, ..., n$, corresponde a uma das n variáveis incógnitas x_i. Cada aresta dirigida no grafo corresponde a uma das m desigualdades que envolvem duas incógnitas.

Utilizando uma linguagem mais formal, dado um sistema $Ax \le b$ de restrições de diferença, o ***grafo de restrição*** correspondente é um grafo dirigido ponderado $G = (V, E)$, onde

$$V = \{v_0, v_1, ..., v_n\}$$

e

$$E = \{(v_i, v_j) : x_j - x_i \le b_k \text{ é uma restrição}\}$$

$$\cup\{(v_0, v_1), (v_0, v_2), (v_0, v_3), ..., (v_0, v_n)\} .$$

O grafo de restrições contém o vértice adicional v_0, como veremos em breve, para garantir que o grafo terá algum vértice que pode alcançar todos os outros vértices. Assim, o conjunto de vértices V consiste em um vértice v_i para cada incógnita x_i, mais um vértice adicional v_0. O conjunto de arestas E contém uma aresta para cada restrição de diferença mais uma aresta (v_0, v_i) para cada incógnita x_i. Se $x_j - x_i \le b_k$ é uma restrição de diferença, então o peso da aresta (v_i, v_j) é $w(v_i, v_j) = b_k$. O peso de cada aresta que sai de v_0 é 0. A Figura 24.8 mostra o grafo de restrição para o sistema (24.3)–(24.10) de restrições de diferença.

O teorema a seguir mostra que podemos encontrar uma solução para um sistema de restrições de diferença determinando os pesos de caminhos mínimos no grafo de restrição correspondente.

Teorema 24.9

Dado um sistema $Ax \le b$ de restrições de diferença, seja $G = (V, E)$ o grafo de restrição correspondente. Se G não contém nenhum ciclo de peso negativo, então

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), ..., \delta(v_0, v_n)) \tag{24.11}$$

é uma solução viável para o sistema. Se G contém um ciclo de peso negativo, não há solução viável para o sistema.

Prova Primeiro mostramos que, se o grafo de restrição não contém nenhum ciclo de peso negativo, a equação (24.11) dá uma solução viável. Considere qualquer aresta $(v_i, v_j) \in E$. Pela desigualdade triangular, $d(v_0, v_j) \leq d(v_0, v_i) + w(v_i, v_j)$ ou, o que é equivalente, $d(v_0, v_j) - d(v_0, v_i) \leq w(v_i, v_j)$. Assim, fazer $x_i = d(v_0, v_i)$ e $x_j = d(v_0, v_j)$ satisfaz a restrição de diferença $x_j - x_i \leq w(v_i, v_j)$ que corresponde à aresta (v_i, v_j).

Agora mostramos que, se o grafo de restrição contém um ciclo de peso negativo, o sistema de restrições de diferença não tem nenhuma solução viável. Sem prejuízo da generalidade, seja o ciclo de peso negativo $c = \langle v_1, v_2, ..., v_k \rangle$, onde $v_1 = v_k$. (O vértice v_0 não pode estar no ciclo c porque não tem nenhuma aresta de entrada.) O ciclo c corresponde às seguintes restrições de diferença:

$$
\begin{aligned}
x_K - x_{K-1} &\leq w(v_1, v_2) , \\
x_3 - x_2 &\leq w(v_2, v_3) , \\
&\vdots \\
x_k - x_{k-1} &\leq w(v_{k-2}, v_{k-1}) , \\
x_1 - x_{k-1} &\leq w(v_{k-1}, v_k) .
\end{aligned}
$$

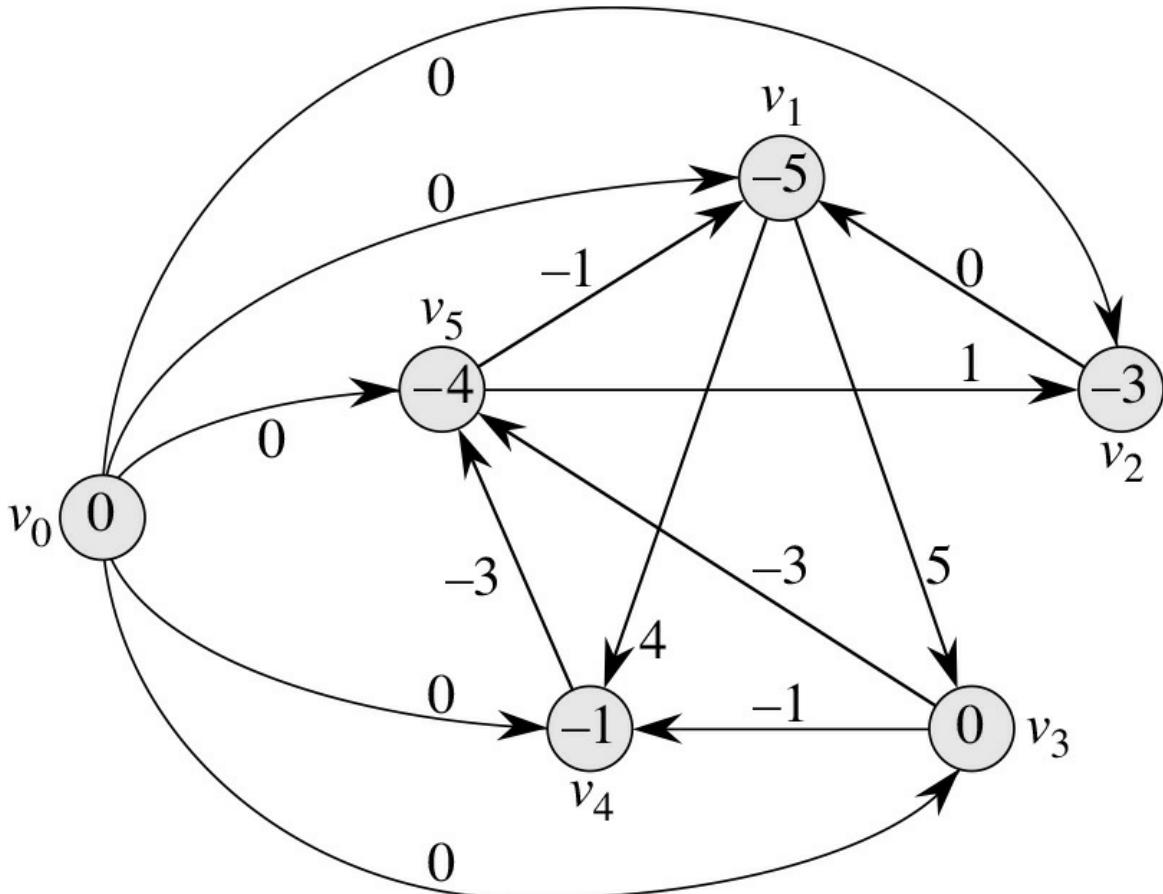



Figura 24.8 Grafo de restrição correspondente ao sistema (24.3)-(24.10) de restrições de diferença. O valor de $d(v_0, v_1)$ aparece em cada vértice v_i. Uma solução viável para o sistema é $x = (-5, -3, 0, -1, -4)$.

Suponha que x tem uma solução que satisfaz cada uma dessas k desigualdades e então deduzimos uma contradição. A solução também deve satisfazer a desigualdade que resulta quando somamos as k desigualdades. Se somarmos os lados esquerdos, cada incógnita x_i é somada uma vez e subtraída uma vez (lembre–se de que $v_1 = v_k$ implica $x_1 = x_k$), de modo que o lado esquerdo da soma é 0. A soma dos valores do lado direito é $w(c)$ e, assim, obtemos $0 \leq w(c)$. Porém, visto que c é um ciclo de peso negativo, $w(c) < 0$, obtemos a contradição $0 \leq w(c) < 0$.

Resolução de sistemas de restrições de diferença

O Teorema 24.9 afirma que podemos usar o algoritmo de BellmanFord para resolver um sistema de restrições de diferenças. Como o grafo de restrição contém arestas que partem do vértice de fonte v_0 para todos os outros vértices, qualquer ciclo de peso negativo no grafo de restrição pode ser alcançado de v_0. Se o algoritmo de BellmanFord retorna TRUE, então os pesos dos caminhos mínimos dão uma solução viável para o sistema. Por exemplo, na Figura 24.8, os pesos de caminhos mínimos dão a solução viável $x = (-5, -3, 0, -1, -4)$ e, pelo Lema 24.8, $x = (d - 5, d - 3, d, d - 1, d - 4)$ também é uma solução viável para qualquer constante d. Se o algoritmo de BellmanFord retorna FALSE, não existe nenhuma solução viável para o sistema de restrições de diferença.

Um sistema de restrições de diferença com m restrições para n incógnitas produz um grafo com $n + 1$ vértices e $n + m$ arestas. Assim, usando o algoritmo de Bellman–Ford, podemos resolver o sistema no tempo $O((n + 1)(n + m)) = O(n_2 + nm)$. O Exercício 24.4–5 pede que você modifique o algoritmo para que seja executado no tempo $O(nm)$, mesmo que m seja muito menor que n.

Exercícios

24.4-1 Encontre uma solução viável ou determine que não existe nenhuma solução viável para o seguinte sistema de restrições:

$$\begin{aligned}
x_1 - x_2 &\leq 1, \\
x_1 - x_4 &\leq -4, \\
x_2 - x_3 &\leq 2, \\
x_2 - x_5 &\leq 7, \\
x_2 - x_6 &\leq 5, \\
x_3 - x_6 &\leq 10, \\
x_4 - x_2 &\leq 2, \\
x_5 - x_1 &\leq -1, \\
x_5 - x_4 &\leq 3, \\
x_6 - x_3 &\leq -8.
\end{aligned}$$

24.4-2 Encontre uma solução viável ou determine que não existe nenhuma solução viável para o seguinte sistema de restrições:

$$x_1 - x_2 \leq 4 ,$$
$$x_1 - x_5 \leq 5 ,$$
$$x_2 - x_4 \leq -6 ,$$
$$x_3 - x_2 \leq 1 ,$$
$$x_4 - x_1 \leq 3 ,$$
$$x_4 - x_3 \leq 5 ,$$
$$x_4 - x_5 \leq 10 ,$$
$$x_5 - x_3 \leq -4 ,$$
$$x_5 - x_4 \leq -8 .$$

24.4-3 Algum peso de caminho mínimo que parte do novo vértice v_0 em um grafo de restrição pode ser positivo? Explique.

24.4-4 Expresse o problema do caminho mínimo para um par como um programa linear.

24.4-5 Mostre como modificar ligeiramente o algoritmo de BellmanFord de modo que, ao usarmos esse algoritmo para resolver um sistema de restrições de diferença com m desigualdades para n incógnitas, o tempo de execução seja $O(nm)$.

24.4-6 Suponha que, além de um sistema de restrições de diferença, queiramos tratar ***restrições de igualdade*** da forma $x_i = x_j + b_k$. Mostre como adaptar o algoritmo de BellmanFord para resolver essa variedade de sistema de restrição.

24.4-7 Mostre como resolver um sistema de restrições de diferença por um algoritmo semelhante ao de BellmanFord em um grafo de restrição sem o vértice extra v_0.

24.4-8 ★ Seja $Ax \leq b$ um sistema de m restrições de diferença para n incógnitas. Mostre que o algoritmo de BellmanFord, quando executado para o grafo de restrição correspondente, maximiza $\sum_{i=1}^{n} x_i$ sujeito a $Ax \leq b$ e $x_i \leq 0$ para todo x_i.

24.4-9 ★ Mostre que o algoritmo de BellmanFord, quando executado para o grafo de restrição para um sistema $Ax \leq b$ de restrições de diferença, minimiza a quantidade $(\max\{x_i\} - \min\{x_i\})$ sujeita a $Ax \leq b$. Explique como esse fato poderia vir a calhar se o algoritmo for usado para programar serviços de construção.

24.4-10 Suponha que toda linha na matriz A de um programa linear $Ax \leq b$ corresponda a uma restrição de diferença, a uma restrição de variável única da forma $x_i \leq b_k$, ou a uma restrição de variável única da forma $-x_i \leq b_k$. Mostre como o algoritmo de BellmanFord pode ser adaptado para resolver essa variedade de sistema de restrições.

24.4-11 Dê um algoritmo eficiente para resolver um sistema $Ax \leq b$ de restrições de diferença quando todos os elementos de b são valores reais e todas as incógnitas x_i devem ser inteiros.

24.4-12 ★Dê um algoritmo eficiente para resolver um sistema $Ax \leq b$ de restrições de diferença quando todos os elementos de b são valores reais e um subconjunto especificado de algumas incógnitas x_i, mas não necessariamente de todas, devem ser inteiros.

24.5 PROVAS DE PROPRIEDADES DE CAMINHOS MÍNIMOS

Em todo este capítulo, nossos argumentos de correção se basearam na desigualdade triangular, na propriedade do limite superior, na propriedade da inexistência de caminho, na propriedade de convergência, na propriedade de relaxamento de caminhos e na propriedade do subgrafo dos predecessores. Enunciamos essas propriedades sem proválas no início deste capítulo. Nesta seção, provamos cada uma delas.

A desigualdade triangular

Quando estudamos a busca em largura (Seção 22.2), provamos no Lema 22.1 uma propriedade simples de distâncias mais curtas em grafos não ponderados. A desigualdade triangular a seguir generaliza a propriedade para grafos ponderados.

Lema 24.10 (Desigualdade triangular)

Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$ e vértice fonte s. Então, para todas as arestas $(u, v) \in E$, temos

$$\delta(s, v) \leq \delta(s, u) + w(u, v) .$$

Prova Suponha que p seja um caminho mínimo da fonte s ao vértice v. Então, p não tem peso maior que qualquer outro caminho de s a v. Especificamente, o caminho p não tem peso maior que o caminho específico que segue um caminho mínimo da fonte s até o vértice u, e depois percorre a aresta (u, v).

O Exercício 24.53 pede que você trate do caso em que não existe nenhum caminho mínimo de s a v.

Efeitos do relaxamento sobre estimativas de caminhos mínimos

O próximo grupo de lemas descreve como as estimativas de caminhos mínimos são afetadas quando executamos uma sequência de etapas de relaxamento nas arestas de um grafo dirigido ponderado que foi inicializado por INITIALIZE-SINGLE-SOURCE.

Lema 24.11 (Propriedade de limite superior)

Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$. Seja $s \in V$ o vértice fonte e considere G inicializado por INITIALIZE-SINGLE-SOURCE(G, s). Então, $v.d \geq$ d(s, v) para todo $v \in V$, e esse invariante é mantido para qualquer sequência de etapas de relaxamento nas arestas de G. Além disso, tão logo $v.d$ alcance seu limite inferior $d(s, v)$, nunca mais muda.

Prova Provamos o invariante $v.d \geq d(s, v)$ para todos os vértices $v \in V$ por indução em relação ao número de etapas de relaxamento.

Para a base, $v.d \geq d(s, v)$ é certamente verdadeiro após inicialização, já que $v.d = \infty$ implica $v.d \geq d(s, v)$ para todo $v \in V - \{s\}$, já que $s.d = 0 \geq d(s, s)$ (observe que $d(s, s) = -\infty$, se s está em um ciclo de peso negativo, e 0, em caso contrário).
Para o passo de indução, considere o relaxamento de uma aresta (u, v). Pela hipótese de indução, $x.d \geq d(s, x)$ para todo $x \in V$ antes do relaxamento. O único valor d que pode mudar é $v.d$. Se ele
mudar, temos

$$
\begin{aligned}
v.d &= u.d + w(u, v) \\
&\geq \delta(s, u) + w(u, v) && \text{(por hipótese de indução)} \\
&\geq \delta(s, v) && \text{(pela desigualdade triangular)} ,
\end{aligned}
$$

e, portanto, o invariante é mantido.

Para ver que o valor de $v.d$ nunca muda depois que $v.d = d(s, v)$, observe que, por ter alcançado seu limite inferior, $v.d$ não pode diminuir porque acabamos de mostrar que $v.d \geq d(s, v)$, e não pode aumentar porque as etapas de relaxamento não aumentam valores de d.

Corolário 24.12 (Propriedade da inexistência de caminho)

Suponha que, em um grafo dirigido ponderado $G = (V, E)$ com função peso $w : E \rightarrow$, nenhum caminho conecte o vértice fonte $s \in V$ a determinado vértice $v \in V$. Então, depois que o grafo é inicializado por INITIALIZE-SINGLE-SOURCE(G, s), temos $v.d = d(s, v) = \infty$, e essa igualdade é mantida como um invariante para qualquer sequência de etapas de relaxamento nas arestas de G.

Prova Pela propriedade de limite superior, temos sempre $\infty = d(s, v) \leq v.d$ e, portanto, $v.d = \infty = d(s, v)$.

Lema 24.13

Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$ e seja $(u, v) \in E$. Então, imediatamente após relaxar a aresta (u, v) pela execução de RELAX(u, v, w), temos $v.d \leq u.d + w(u, v)$.

Prova Se, imediatamente antes de relaxar a aresta (u, v), temos $v.d > u.d + w(u, v)$, então $v.d = u.d + w(u, v)$ daí em diante. Se, em vez disso, $v.d \leq u.d + w(u, v)$ imediatamente antes do relaxamento, então nem $d[u]$ nem $d[v]$ se alteram e, assim, $v.d \leq u.d + w(u, v)$ daí em diante.

Lema 24.14 (Propriedade de convergência)

Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$, seja $s \in V$ um vértice de fonte, e seja s $u \rightarrow v$ um caminho mínimo em G para alguns vértices u, $v \in V$. Suponha que G seja inicializado por INITIALIZE-SINGLE-SOURCE(G, s) e depois uma sequência de etapas de relaxamento que inclua a chamada RELAX(u, v, w) é executada para as arestas de G. Se $u.d = d(s, u)$ em qualquer tempo anterior à chamada, então $u.d = d(s, v)$ em todos os tempos após a chamada.

Prova Pela propriedade do limite superior, se $u.d = d(s, u)$ em algum ponto antes do relaxamento da aresta (u, v), então essa igualdade se mantém válida daí em diante. Em particular, após o relaxamento da aresta (u, v), temos

$$
\begin{aligned}
v.d &\leq u.d + w(u, v) && \text{(pelo Lema 24.13)} \\
&= \delta(s, u) + w(u, v) \\
&= \delta(s, v) && \text{(pelo Lema 24.1)} .
\end{aligned}
$$

Pela propriedade do limite superior, $v.d \geq d(s, v)$, da qual concluímos que $v.d = d(s, v)$, e essa igualdade é mantida daí em diante.

Lema 24.15 (Propriedade de relaxamento de caminho)

Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$ e seja $s \in V$ um vértice de fonte. Considere qualquer caminho mínimo $p = \langle v_0, v_1, ..., v_k \rangle$ de $s = v_0$ a v_k. Se G é inicializado por INITIALIZE-SINGLE-SOURCE(G, s) e depois ocorre uma sequência de etapas de relaxamento que inclui, pela ordem, relaxar as arestas (v_0, v_1), (v_1, v_2), ...,

(v_k–1, v_k), então $v_k.d = d(s, v_k)$ depois desses relaxamentos e todas as vezes daí em diante. Essa propriedade se mantém válida, não importando quais outros relaxamentos de arestas ocorram, inclusive relaxamentos entremeados com relaxamentos das arestas de p.

Prova Mostramos por indução que depois que a $i$ésima aresta do caminho p é relaxada, temos $v_i.d = d(s, v_i)$. Para a base, $i = 0$, e, antes que quaisquer arestas de p tenham sido relaxadas, temos pela inicialização que $v_0.d = s.d = 0 = d(s, s)$. Pela propriedade do limite superior, o valor de $d[s]$ nunca muda após a inicialização.
Para o passo de indução, supomos que $v_{i-1}.d = d(s, v_{i-1})$ e examinamos o que acontece quando relaxamos a aresta (v_{i-1}, v_i). Pela propriedade de convergência, após o relaxamento dessa aresta, temos $v_i.d = d(s, v_i)$, e essa igualdade é mantida todas as vezes depois disso.

Relaxamento e árvores de caminhos mínimos

Agora mostramos que, tão logo uma sequência de relaxamentos tenha provocado a convergência de estimativas de caminhos mínimos para pesos de caminhos mínimos, o subgrafo predecessor G_p induzido pelos valores de p resultantes é uma árvore de caminhos mínimos para G. Começamos com o lema a seguir, que mostra que o subgrafo dos predecessores sempre forma uma árvore enraizada cuja raiz é a fonte.

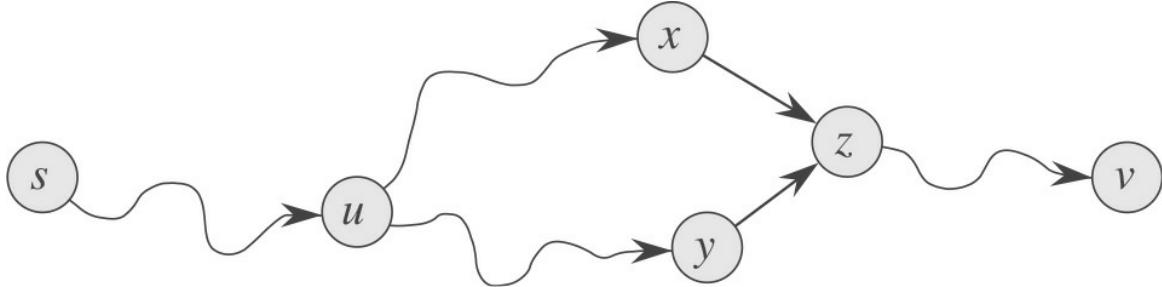
Lema 24.16

Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$, seja $s \in V$ um vértice de fonte e suponha que G não contenha nenhum ciclo de peso negativo que possa ser alcançado de s. Então, depois que o grafo é inicializado por INITIALIZE-SINGLE-SOURCE(G, s), o subgrafo dos predecessores G_p forma uma árvore enraizada com raiz s, e qualquer sequência de etapas de relaxamento em arestas de G mantém essa propriedade como um invariante.

Prova Inicialmente, o único vértice em G_p é o vértice de fonte, e o lema é trivialmente verdadeiro. Considere um subgrafo dos predecessores G_p que surja após uma sequência de etapas de relaxamento. Primeiro, provaremos que G_p é acíclico. Suponha, por contradição, que alguma etapa de relaxamento crie um ciclo no grafo G_p. Seja $c = \langle v_0, v_1, ..., v_k \rangle$ o ciclo onde $v_k = v_0$. Então, $v_i.p = v_{i-1}$ para $i = 1, 2, ..., k$ e, sem prejuízo da generalidade, podemos supor que o relaxamento da aresta (v_{k-1}, v_k) criou o ciclo em G_p.

Afirmamos que todos os vértices no ciclo c podem ser alcançados da fonte s. Por quê? Cada vértice em c tem um predecessor não NIL; portanto, uma estimativa de caminho mínimo finito foi atribuída a cada vértice c quando lhe foi atribuído seu valor p não NIL. Pela propriedade do limite superior, cada vértice no ciclo c tem um peso de caminho mínimo finito, o que implica que ele pode ser alcançado de s.

Examinaremos as estimativas de caminhos mínimos em c imediatamente antes da chamada RELAX(v_{k-1}, v_k, w) e mostraremos que c é um ciclo de peso negativo, contradizendo assim a hipótese de que G não contém nenhum ciclo de peso negativo que possa ser alcançado da fonte. Imediatamente antes da chamada, temos $v_i.p = v_{i-1}$ para $i = 1, 2, ..., k - 1$. Assim, para $i = 1, 2, ..., k - 1$, a última atualização para $v_i.d$ foi feita pela atribuição $v_i.d = v_{i-1}.d + w(v_{i-1}, v_i)$. Se $v_{i-1}.d$ mudou desde então, ela diminuiu. Por essa razão, imediatamente antes da chamada a RELAX(v_{k-1}, v_k, w), temos

$$v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i) \qquad \text{para todo } i = 1, 2, ..., k-1. \qquad (24.12)$$

Como $v_k.p$ é alterado pela chamada, imediatamente antes temos também a desigualdade estrita

$$v_k.d > v_{k-1}.d + w(v_{k-1}, v_k).$$

Somando essa desigualdade estrita com as $k - 1$ desigualdades (24.12), obtemos a soma das estimativas dos caminhos mínimos em torno do ciclo *c:*

$$\sum_{i=1}^{k} v_i.d \; > \; \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)),$$
$$= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Figura 24.9 A figura mostra que um caminho simples em G_p da fonte *s* ao vértice *v* é único. Se houvesse dois caminhos p_1 ($s \; u \; x \rightarrow z \; v$) e p_2 ($s \; u \; y \rightarrow z \; v$), onde $x \neq y$, então $z.p = x$ e $z.p = y$, uma contradição.

Mas,

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d \; ,$$

já que cada vértice no ciclo *c* aparece exatamente uma vez em cada somatório. Essa igualdade implica

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Assim, a soma dos pesos em torno do ciclo *c* é negativa, o que dá a contradição desejada.

Agora já provamos que G_p é um grafo acíclico dirigido. Para mostrar que ele forma uma árvore enraizada com raiz *s*, basta provar (veja o Exercício B.5–2) que, para cada vértice $v \in V_p$, há um único caminho simples de *s* a *v* em G_p.

Primeiro, devemos mostrar que existe um caminho de *s* a cada vértice em V_p. Os vértices em V_p são os que têm valores *p* não NIL mais *s*. Aqui, a ideia é provar por indução que existe um caminho de *s* a todos os vértices em V_p. Os detalhes ficam para o Exercício 24.56.

Para concluir a prova do lema, devemos mostrar agora que, para qualquer vértice $v \in V_p$, o grafo G_p contém no máximo um caminho simples de *s* a *v*. Suponha o contrário. Isto é, suponha que, como mostra a Figura 24.9, G_p contenha dois caminhos simples de *s* a algum vértice $v.p_1$, que decompomos em $s \; u \; x \rightarrow z \; v$, e p_2, que decompomos em $s \; u \; y \rightarrow z \; v$, onde $x \neq y$ se bem que *u* poderia ser *s* e *z* poderia ser *v*). Mas, então, $z.p = x$ e $z.p = y$, o que implica a contradição $x = y$. Concluímos que G_p contém um caminho simples único de *s* a *v* e, assim, G_p forma uma árvore enraizada com raiz *s*.

Agora podemos mostrar que, se depois de executarmos uma sequência de etapas de relaxamento todos os vértices tiverem recebido a atribuição de seus pesos de caminhos mínimos verdadeiros, o subgrafo dos predecessores G_p será

uma árvore de caminhos mínimos.

Lema 24.17 (Propriedade do subgrafos predecessores)

Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$, seja $s \in V$ um vértice fonte, e suponha que G não contenha nenhum ciclo de peso negativo que possa ser alcançado de s. Vamos chamar INITIALIZE-SINGLE-SOURCE(G, s) e depois executar qualquer sequência de etapas de relaxamento para arestas de G que produza $v.d = d(s, v)$ para todo $v \in V$. Então, o subgrafo predecessor G_p é uma árvore de caminhos mínimos com raiz em s.

Prova Devemos provar que as três propriedades de árvores de caminhos mínimos dadas na página 647 são válidas para G_p. Para ilustrar a primeira propriedade, devemos mostrar que V_p é o conjunto de vértices que pode ser alcançado de s. Por definição, um peso de caminho mínimo $d(s, v)$ é finito se e somente se v pode ser alcançado de s e, portanto, os vértices que podem ser alcançados de s são exatamente aqueles que têm valores de d finitos. Porém, um vértice $v \in V - \{s\}$ recebeu a atribuição de um valor finito para $v.d$ se e somente se $v.p \neq$ NIL. Assim, os vértices em V_p são exatamente aqueles que podem ser alcançados de s.

A segunda propriedade decorre diretamente do Lema 24.16.

Portanto, resta provar a última propriedade de árvores de caminhos mínimos: para cada vértice $v \in V_p$, o único caminho simples s v em G_p é um caminho mínimo de s a v em G. Seja $p = \langle v_0, v_1, ..., v_k \rangle$, onde $v_0 = s$ e $v_k = v$. Para $i = 1, 2, ..., k$, temos $v.d = d(s, v_i)$ e também $v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i)$, do que concluímos $w(v_{i-1}, v_i) \leq d(s, v_i) - d(s, v_{i-1})$. A soma dos pesos ao longo do caminho p produz

$$\begin{aligned} w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) && \text{(porque a soma é telescópica)} \\ &= \delta(s, v_k) && \text{(porque } \delta(s, v_0) = \delta(s, s) = 0) . \end{aligned}$$

Assim, $w(p) \leq d(s, v_k)$. Visto que $d(s, v_k)$ é um limite inferior para o peso de qualquer caminho de s a v_k, concluímos que $w(p) = d(s, v_k)$ e, portanto, p é um caminho mínimo de s a $v = v_k$.

Exercícios

24.5-1 Dê duas árvores de caminhos mínimos para o grafo dirigido da Figura 24.2, além das duas mostradas.

24.5-2 Dê um exemplo de grafo dirigido ponderado $G = (V, E)$ com função peso $w : E \rightarrow$ e vértice fonte s tal que G satisfaça a seguinte propriedade: para toda aresta $(u, v) \in E$ existe uma árvore de caminhos mínimos com raiz em s que contém (u, v) e outra árvore de caminhos mínimos com raiz em s que não contém (u, v).

24.5-3 Aperfeiçoe a prova do Lema 24.10 para tratar de casos nos quais os pesos de caminhos mínimos são ∞ ou $-\infty$.

24.5-4 Seja $G = (V, E)$ um grafo dirigido ponderado com vértice fonte s e suponha G inicializado por INITIALIZE-SINGLE-SOURCE(G, s). Prove que, se uma sequência de etapas de relaxamento define $s.p$ com um valor não NIL, então G contém um ciclo de peso negativo.

24.5-5 Seja $G = (V, E)$ um grafo dirigido ponderado sem arestas de peso negativo. Seja $s \in V$ o vértice fonte e suponha que permitimos que $v.p$ seja o predecessor de v em *algum* caminho mínimo de v até a fonte s se $v \in V - \{s\}$ puder ser alcançado de s, e NIL em caso contrário. Dê um exemplo de tal grafo G e de uma atribuição de valores de p que produza um ciclo em G_p. (Pelo Lema 24.16, tal atribuição não pode ser produzida por uma sequência de etapas de relaxamento.)

24.5-6 Seja $G = (V, E)$ um grafo dirigido ponderado com função peso $w : E \rightarrow$ e sem ciclos de peso negativo. Seja $s \in V$ o vértice fonte e suponha G inicializado por INITIALIZE-SINGLE- SOURCE(G, s). Prove que, para todo vértice $v \in V_p$, existe um caminho de s a v em G_p e que essa propriedade é mantida como um invariante para qualquer sequência de relaxamentos.

24.5-7 Seja $G = (V, E)$ um grafo dirigido ponderado que não contém nenhum ciclo de peso negativo. Seja $s \in V$ o vértice fonte e suponha G inicializado por INITIALIZE-SINGLE-SOURCE(G, s). Prove que existe uma sequência de $|V| - 1$ etapas de relaxamento que produz $v.d = d(s, v)$ para todo $v \in V$.

24.5-8 Seja G um grafo dirigido ponderado arbitrário com um ciclo de peso negativo que pode ser alcançado do vértice de fonte s. Mostre como construir uma sequência infinita de relaxamentos das arestas de G tal que todo relaxamento provoque mudança em uma estimativa de caminho mínimo.

Problemas

24-1 *Aperfeiçoamento de Yen para BellmanFord*

Suponha que ordenamos os relaxamentos de arestas em cada passagem do algoritmo de BellmanFord da seguinte maneira: antes da primeira passagem atribuímos uma ordem linear arbitrária $v_1, v_2, ..., v_{|V|}$ aos vértices do grafo de entrada $G = (V, E)$. Então, particionamos o conjunto de arestas E em $E_f \cup E_b$, onde $E_f = \{(v_i, v_j) \in E : i < j\}$ e $E_b = \{(v_i, v_j) \in E : i > j\}$. (Suponha que G não contenha nenhum laço, de modo que toda aresta está em E_f ou em E_b.) Defina $G_f = (V, E_f)$ e $G_b = (V, E_b)$.

a. Prove que G_f é acíclico com ordenação topológica $\langle v_1, v_2, ..., v_{|V|}\rangle$ e que G_b é acíclico com ordenação topológica $\langle v_{|V|}, v_{|V|-1} ..., v_1\rangle$.

Suponha que implementemos cada passagem do algoritmo de BellmanFord da maneira descrita a seguir. Visitamos cada vértice na ordem $v_1, v_2, ..., v_{|V|}$, relaxando as arestas de E_f que partem do vértice. Então, visitamos cada vértice na ordem $v_{|V|}, v_{|V|-1}, ..., v_1$, relaxando as arestas de E_b que partem do vértice.

b. Prove que, com esse esquema, se G não contém nenhum ciclo de peso negativo que possa ser alcançado do vértice de fonte s, então, depois de apenas $|V|/2$ passagens pelas arestas, $v.d = d(s, v)$ para todos os vértices $v \in V$.

c. Esse esquema melhora o tempo de execução assintótico do algoritmo de BellmanFord?

24-2 *Aninhamento de caixas*

Uma caixa com d dimensões $(x_1, x_2, ..., x_d)$ se ***aninha*** dentro de outra caixa com dimensões $(y_1, y_2, ..., y_d)$ se existe uma permutação p para $\{1, 2, ..., d\}$ tal que $x_{p(1)} < y_1, x_{p(2)} < y_2, ..., x_{p(d)} < y_d$.

a. Demonstre que a relação de aninhamento é transitiva.

b. Descreva um método eficiente para determinar se uma caixa com d dimensões se aninha ou não dentro de outra.

c. Suponha que você recebeu um conjunto de n caixas com d dimensões $\{B_1, B_2, ..., B_n\}$. Dê um algoritmo eficiente para determinar a sequência mais longa de $\langle B_{i_1}, B_{i_2}, ..., B_{i_k} \rangle$ caixas tal que B_{i_j} fique aninhada dentro de $B_{i_{j+1}}$ para $j = 1, 2, ..., k - 1$. Expresse o tempo de execução do seu algoritmo em termos de n e d.

24-3 Arbitragem

A ***arbitragem*** é a utilização de discrepâncias em taxas de câmbio para transformar uma unidade de uma moeda em mais de uma unidade da mesma moeda. Por exemplo, suponha que 1 dólar americano compre 49 rupias indianas. Uma rupia indiana compra dois ienes japoneses, e um iene japonês compra 0,0107 dólar americano. Então, convertendo moedas, um comerciante pode começar com um dólar americano e comprar $49 \cdot 2 \cdot 0{,}0107 = 1{,}0486$ dólar americano, obtendo assim um lucro de 4,86%.

Suponha que recebemos n moedas $c_1, c_2, ..., c_n$ e uma tabela R $n \times n$ de taxas de câmbio, tal que uma unidade da moeda c_i compre $R[i, j]$ unidades da moeda c_j.

a. Dê um algoritmo eficiente para determinar se existe ou não uma sequência de moedas $\langle c_{i_1}, c_{i_2}, ..., c_{i_k} \rangle$ tal que

$$R[i_1, i_2] \times R[i_2, i_3] \dots R[i_{k-1}, i_k] \times R[i_k, i_1] > 1 .$$

Analise o tempo de execução do seu algoritmo.

b. Dê um algoritmo eficiente para imprimir tal sequência, se ela existir. Analise o tempo de execução do seu algoritmo.

24-4 Algoritmo de mudança de escala de Gabow para caminhos mínimos de fonte única

O algoritmo de ***mudança de escala*** resolve um problema considerando inicialmente apenas o bit de ordem mais alta de cada valor de entrada relevante (como um peso de aresta). Em seguida, refina a solução inicial observando os dois bits de ordem mais alta. Prossegue examinado cada vez mais bits de ordem mais alta e refinando a solução toda vez, até ter examinado todos os bits e calculado a solução correta.

Nesse problema, examinamos um algoritmo para calcular os caminhos mínimos de uma fonte única por escalonamento de pesos de arestas. Temos um grafo dirigido $G = (V, E)$ com pesos de arestas inteiros não negativos w. Seja $W = \max_{(u, v) \in E} \{w(u, v)\}$. Nossa meta é desenvolver um algoritmo que seja executado no tempo $O(E \lg W)$. Supomos que todos os vértices podem ser alcançados da fonte.

O algoritmo descobre os bits na representação binária dos pesos de arestas um por vez, desde o bit mais significativo até o bit menos significativo. Especificamente, seja $k = \lg(W + 1)$ o número de bits na representação binária de W e, para $i = 1, 2, ..., k$, seja $w_i(u, v) = w(u, v)/2^{k-i}$. Isto é, $w_i(u, v)$ é a versão em "escala reduzida" de $w(u, v)$ dada pelos i bits mais significativos de $w(u, v)$. (Assim, $w_k(u, v) = w(u, v)$ para todo $(u, v) \in E$.) Por exemplo, se $k = 5$ e $w(u, v) = 25$, cuja representação binária é $\langle 11001 \rangle$, então $w_3(u, v) = \langle 110 \rangle = 6$. Como outro exemplo com $k = 5$, se $w(u, v) = \langle 00100 \rangle = 4$, então $w_3(u, v) = \langle 001 \rangle = 1$. Vamos definir $d^i(u, v)$ como o peso do caminho mínimo do vértice u ao vértice v utilizando a função peso w_i. Assim, $d^k(u, v) = d(u, v)$ para todo $u, v \in V$. Para dado vértice fonte s, o algoritmo de escalonamento calcula primeiro os pesos de caminhos mínimos $d^1(s, v)$ para todo $v \in V$, depois calcula $d^2(s, v)$ para todo $v \in V$, e assim por diante, até calcular $d^k(s, v)$ para todo $v \in V$. Supomos em todo esse processo que $|E| \geq$

$|V| - 1$, e veremos que o cálculo de d_i por d_{i-1} demora o tempo $O(E)$, de modo que o algoritmo inteiro demora o tempo $O(kE) = O(E \lg W)$.

a. Suponha que, para todos os vértices $v \in V$, temos $d(s, v) \leq |E|$. Mostre que podemos calcular $d(s, v)$ para todo $v \in V$ no tempo $O(E)$.

b. Mostre que podemos calcular $d_1(s, v)$ para todo $v \in V$ no tempo $O(E)$.

Agora, vamos focalizar o cálculo de d_i a partir de d_{i-1}.

c. Prove que, para $i = 2, 3, ..., k$, temos $w_i(u, v) = 2w_{i-1}(u, v)$ ou $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Em seguida, prove que

$$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$$

para todo $v \in V$.

d. Defina, para $i = 2, 3, ..., k$ e para todo $(u, v) \in E$,

$$\hat{w}_i(u, v) = w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v) .$$

Prove que, para $i = 2, 3, ..., k$ e todo $u, v \in V$, o valor "reponderado" (u, v) da aresta (u, v) é um inteiro não negativo.

e. Agora, defina $\hat{\delta}_i(s, v)$ como o peso do caminho mínimo de s a v usando a função peso $\hat{w}_i$. Prove que, para $i = 2, 3, ..., k$ e todo $v \in V$,

$$\delta_i(s, v) = \hat{\delta}_i(s, v) + 2\delta_{i-1}(s, v)$$

e que $\hat{\delta}(s, v) \leq |E|$.

f. Mostre como calcular $d_i(s, v)$ a partir de $d_{i-1}(s, v)$ para todo $v \in V$ no tempo $O(E)$ e conclua que podemos calcular $d(s, v)$ para todo $v \in V$ no tempo $O(E \lg W)$.

24-5 Algoritmo do ciclo de peso médio mínimo de Karp

Seja $G = (V, E)$ um grafo dirigido com função peso $w : E \rightarrow$ e seja $n = |V|$. Definimos o ***peso médio*** de um ciclo $c = \langle e_1, e_2, ..., e_k \rangle$ de arestas em E como

$$\mu(c) = \frac{1}{k}\sum_{i=1}^{k} w(e_i).$$

Seja $m^* = \min_c m(c)$, onde c varia em todos os ciclos dirigidos em G. Denominamos ***ciclo de peso médio mínimo*** um ciclo c para o qual $m(c) = m^*$. Este problema investiga um algoritmo eficiente para cálculo de m^*.

Suponha, sem prejuízo da generalidade, que todo vértice $v \in V$ pode ser alcançado de um vértice fonte $s \in V$. Seja $d(s, v)$ o peso de um caminho mínimo de s a v e seja $d^k(s, v)$ o peso de um caminho mínimo de s a v consistindo em *exatamente* k arestas. Se não existe nenhum caminho de s a v com exatamente k arestas, então $d^k(s, v) = \infty$.

a. Mostre que, se $m^* = 0$, G não contém nenhum ciclo de peso negativo e $d(s, v) = \min_{0 \leq k \leq n-1} \delta_k(s, v)$ para todos os vértices $v \in V$.

b. Mostre que, se $m^* = 0$, então

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} \geq 0$$

para todos os vértices $v \in V$. (*Sugestão*: Use ambas as propriedades da parte (a).)

c. Seja c um ciclo de peso 0, e sejam u e v quaisquer dois vértices em c. Suponha que $m^* = 0$ e que o peso do caminho simples de u a v ao longo do ciclo seja x. Prove que $d(s, v) = d(s, u) + x$. (*Sugestão*: O peso do caminho simples de v a u ao longo do ciclo é $-x$.)

d. Mostre que, se $m^* = 0$, então em cada ciclo de peso médio mínimo existe um vértice v tal que

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0.$$

(*Sugestão*: Mostre como estender um caminho mínimo até qualquer vértice em um ciclo de peso médio mínimo ao longo do ciclo para formar um caminho mínimo até o próximo vértice no ciclo.)

e. Mostre que, se $m^* = 0$, então

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k} = 0.$$

f. Mostre que, se adicionarmos uma constante t ao peso de cada aresta de G, m^* aumenta de t. Use esse fato para mostrar que

$$\mu^* = \min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k}.$$

g. Dê um algoritmo de tempo $O(V E)$ para calcular m^*.

24-6 Caminhos mínimos bitônicos

Uma sequência é ***bitônica*** se cresce monotonicamente e depois decresce monotonicamente ou se, por um deslocamento circular, cresce monotonicamente e depois decresce monotonicamente. Por exemplo, as sequências $\langle 1, 4, 6, 8, 3, -2 \rangle$, $\langle 9, 2, -4, -10, -5 \rangle$ e $\langle 1, 2, 3, 4 \rangle$ são bitônicas, mas $\langle 1, 3, 12, 4, 2, 10 \rangle$ não é bitônica. (Veja o Problema 15–3, que aborda o problema do caixeiro–viajante euclidiano bitônico.)

Suponha que tenhamos um grafo dirigido $G = (V, E)$ com função peso $w : E \rightarrow$ onde todos os pesos de arestas são únicos e desejamos encontrar caminhos mínimos de fonte única que parta m de um vértice de fonte s. Temos uma informação adicional: para cada vértice $v \in V$, os pesos das arestas ao longo de qualquer caminho mínimo de s a v formam uma sequência bitônica.

Dê o algoritmo mais eficiente que puder para resolver esse problema e analise seu tempo de execução.

NOTAS DO CAPÍTULO

O algoritmo de Dijkstra [88] surgiu em 1959, mas não fazia nenhuma menção a uma fila de prioridades. O algoritmo de BellmanFord se baseia em algoritmos separados criados por Bellman [38] e Ford [109]. Bellman descreve a relação entre caminhos mínimos e restrições de diferença. Lawler [224] descreve o algoritmo de tempo linear para caminhos mínimos em um gad, que ele considera parte do folclore.

Quando os pesos de arestas são inteiros não negativos relativamente pequenos, temos algoritmos mais eficientes para resolver o problema de caminhos mínimos de fonte única. A sequência de valores devolvidos pelas chamadas EXTRACT-MIN no algoritmo de Dijkstra cresce monotonicamente com o tempo. Como discutimos nas notas do Capítulo 6, nesse caso existem várias estruturas de dados que podem implementar as várias operações de filas de prioridades com mais eficiência que um heap binário ou um heap de Fibonacci. Ahuja, Mehlhorn, Orlin e Tarjan [8] dão um algoritmo que é executado no tempo $O(E + V \sqrt{\lg W})$ para grafos com pesos de arestas não negativos, onde W é o maior peso de qualquer aresta no grafo. Os melhores limites são dados por Thorup [337], que dá um algoritmo que é executado no tempo $O(E \lg \lg V)$, e por Raman [291], que dá um algoritmo que é executado no tempo $O(E + V \min \{(\lg V)^{1/3+\in}, (\lg W)^{1/4+\in}\})$. Esses dois algoritmos usam uma quantidade de espaço que depende do tamanho da palavra da máquina subjacente. Embora a quantidade de espaço utilizada possa ser ilimitada no que se refere ao tamanho da entrada, ela pode ser reduzida a linear em relação ao tamanho da entrada com a utilização de hashing aleatorizado.

Para grafos não dirigidos com pesos inteiros, Thorup [336] dá um algoritmo de tempo $O(V + E)$ para caminhos mínimos de fonte única. Ao contrário dos algoritmos mencionados no parágrafo anterior, esse algoritmo não é uma implementação do algoritmo de Dijkstra, já que a sequência de valores retornados por chamadas EXTRACT-MIN não cresce monotonicamente com o tempo.

Para grafos com pesos de arestas negativos, um algoritmo criado por Gabow e Tarjan [122] é executado no tempo $O(\sqrt{V}E \lg(V W))$, e um outro criado por Goldberg [137] é executado no tempo $O(\sqrt{V} E \lg W)$, onde $W = \max_{(u, v) \in E} \{|w(u, v)|\}$.

Cherkassky, Goldberg e Radzik 64 realizaram extensivos experimentos comparando vários algoritmos de caminhos mínimos.

---

[1] Pode parecer estranho que o termo "relaxamento" seja usado para uma operação que restringe um limite superior. O uso do termo é histórico. A saída de uma etapa de relaxamento pode ser vista como um relaxamento da restrição $v.d \leq u.d + w(u, v)$ que, pela desigualdade triangular (Lema 24.10), deve ser satisfeita se $u.d = d(s, u)$ e $v.d = d(s, v)$. Isto é, se $v.d \leq u.d + w(u, v)$, não há nenhuma "pressão" para satisfazer essa restrição e, assim, a restrição é "relaxada".

[2] "PERT" é um acrônimo para "*program evaluation and review technique*" (técnica de avaliação e revisão de programação).

25 CAMINHOS MÍNIMOS ENTRE TODOS OS PARES

Neste capítulo, consideraremos o problema de encontrar caminhos mínimos entre todos os pares de vértices em um grafo. Esse problema poderia surgir na elaboração de uma tabela de distâncias entre todos os pares de cidades em um atlas rodoviário. Como no Capítulo 24, temos um grafo orientado ponderado $G = (V, E)$ com uma função peso w : $E \rightarrow$ que mapeia arestas para pesos de valores reais. Desejamos encontrar, para todos os pares de vértices u, $v \in V$, um caminho mínimo (de peso mínimo) de u a v, onde o peso de um caminho é a soma dos pesos das arestas que o constituem. Normalmente, queremos a saída em forma tabular: a entrada na linha de u e na coluna de v deve ser o peso de um caminho mínimo de u a v.

Podemos resolver um problema de caminhos mínimos para todos os pares executando um algoritmo de caminhos mínimos de fonte única $|V|$ vezes, cada uma dessas vezes considerando um vértice como fonte. Se todos os pesos de arestas são não negativos, podemos usar o algoritmo de Dijkstra. Se usarmos a implementação da fila de prioridade mínima por arranjo linear, o tempo de execução será $O(V_3 + V E) = O(V_3)$. A implementação da fila de prioridade mínima por heap binário mínimo produz um tempo de execução $O(V E \lg V)$, que é uma melhoria se o grafo é esparso. Como alternativa, podemos implementar a fila de prioridade mínima com um heap de Fibonacci, o que produz um tempo de execução $O(V_2 \lg V + VE)$.

Se o grafo tiver arestas de peso negativo não podemos usar o algoritmo de Dijkstra. Em vez disso, temos de executar o algoritmo de Bellman-Ford, mais lento, uma vez para cada vértice. O tempo de execução resultante é $O(V_2E)$ que, em um grafo denso, é $O(V_4)$. Neste capítulo, veremos como podemos obter resultados melhores. Também investigaremos a relação entre o problema de caminhos mínimos para todos os pares e a multiplicação de matrizes, e estudaremos sua estrutura algébrica.

Diferentemente dos algoritmos de fonte única, que consideram uma representação do grafo por lista de adjacências, a maioria dos algoritmos neste capítulo utiliza uma representação por matriz de adjacências. (O algoritmo de Johnson para grafos esparsos, na Seção 25.3, usa listas de adjacências.) Por conveniência, suporemos que os vértices estão numerados como 1, 2, ..., $|V|$, de modo que a entrada é uma matriz $n \times n$ W que representa os pesos de arestas de um grafo dirigido de n vértices $G = (V, E)$. Isto é, $W = (w_{ij})$, onde

$$w_{ij} = \begin{cases} 0 & \text{se } i = j, \\ \text{o peso da aresta dirigida } (i, j) & \text{se } i \neq j \text{ e } (i, j) \in E, \\ \infty & \text{se } i \neq j \text{ e } (i, j) \notin E. \end{cases} \quad (25.1)$$

Permitimos arestas de peso negativo, mas por enquanto supomos que o grafo de entrada não contém nenhum ciclo de peso negativo.

A saída tabular dos algoritmos de caminhos mínimos para todos os pares apresentados neste capítulo é uma matriz $n \times n$ $D = (d_{ij})$, onde a entrada d_{ij} contém o peso de um caminho mínimo do vértice i ao vértice j. Isto é, se $d(i, j)$ representa o peso do caminho mínimo do vértice i ao vértice j (como no Capítulo 24), então $d_{ij} = d(i, j)$ no término.

Para resolver o problema de caminhos mínimos para todos os pares para uma matriz de adjacências de entrada, precisamos calcular não só os pesos de caminhos mínimos, mas também uma ***matriz de predecessores*** P = (p_{ij}), onde p_{ij} é NIL se $i = j$ ou se não existe nenhum caminho de i a j e, caso contrário, p_{ij} é o predecessor de j em um caminho mínimo que parte de i. Exatamente como o subgrafo dos predecessores G_p do Capítulo 24 é uma árvore de caminhos mínimos para um dado vértice fonte, o subgrafo induzido pela i-ésima linha da matriz P deve ser uma árvore de caminhos mínimos com raiz i. Para cada vértice $i \in V$, definimos o ***subgrafo dos predecessores*** de G para i como $G_{p, i} = (V_{p, i}, E_{p, i})$, onde

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

e

$$E_{\pi,i} = \{(\pi_{ij}, j)) : j \in V_{\pi,i} - \{i\}\} .$$

Se $G_{p, i}$ é uma árvore de caminhos mínimos, então o procedimento a seguir, que é uma versão modificada do procedimento PRINT-PATH do Capítulo 22, imprime um caminho mínimo do vértice i ao vértice j.

```
PRINT-ALL-PAIRS-SHORTEST-PATH (Π, i, j)
1  if i = j
2     print i
3  else if π_ij == NIL
4     print "não existe nenhum caminho de " i " para " j
5  else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6     print j
```

Para destacar as características essenciais dos algoritmos para todos os pares neste capítulo, não dedicaremos tanta atenção e espaço à criação e às propriedades de matrizes de predecessores quanto dedicamos aos subgrafos dos predecessores no Capítulo 24. Alguns dos exercícios abordam os aspectos básicos.

Esboço do capítulo

A Seção 25.1 apresenta um algoritmo de programação dinâmica baseado em multiplicação de matrizes para resolver o problema de caminhos mínimos para todos os pares. Usando a técnica de "elevação ao quadrado repetida", podemos conseguir um tempo de execução Q(V_3 lg V). A Seção 25.2 dá um outro algoritmo de programação dinâmica, o algoritmo de Floyd-Warshall, que é executado no tempo Q(V_3). A Seção 25.2 aborda também o problema de encontrar o fecho transitivo de um grafo dirigido, que está relacionado com o problema de caminhos mínimos para todos os pares. Finalmente, a Seção 25.3 apresenta o algoritmo de Johnson, que resolve o problema dos caminhos mínimos para todos os pares no tempo $O(V_2 \lg V + V E)$ e é uma boa escolha para grafos grandes e esparsos.

Antes de prosseguir, precisamos estabelecer algumas convenções para representações por matrizes de adjacências. Em primeiro lugar, em geral, suporemos que o grafo de entrada $G = (V, E)$ tem n vértices, de modo que $n = |V|$. Em segundo lugar, usaremos a convenção de denotar matrizes por letras maiúsculas, como W, L ou D, e seus elementos individuais por letras minúsculas indexadas, como w_{ij}, l_{ij} ou d_{ij}. Algumas matrizes terão índices entre parênteses, como em $L^{(m)} = \left(l_{ij}^{(m)}\right)$ ou $D^{(m)} = \left(d_{ij}^{(m)}\right)$, para indicar iterados. Por fim, para uma matriz $n \times n$ A dada, supomos que o valor de n está armazenado no atributo $A.linhas$.

25.1 CAMINHOS MÍNIMOS E MULTIPLICAÇÃO DE MATRIZES

Esta seção apresenta um algoritmo de programação dinâmica para o problema de caminhos mínimos para todos os pares em um grafo dirigido $G = (V, E)$. Cada laço principal do programa dinâmico invocará uma operação que é muito semelhante à multiplicação de matrizes, de modo que o algoritmo será parecido com uma multiplicação de matrizes repetida. Começaremos desenvolvendo um algoritmo de tempo $O(V_4)$ para o problema de caminhos mínimos para todos os pares e depois melhoraremos seu tempo de execução para $O(V_3 \lg V)$.

Antes de continuar, vamos recapitular rapidamente as etapas para desenvolver um algoritmo de programação dinâmica dadas no Capítulo 15.

1. Caracterizar a estrutura de uma solução ótima.
2. Definir recursivamente o valor de uma solução ótima.
3. Calcular o valor de uma solução ótima de baixo para cima.

Reservamos a quarta etapa — construção de uma solução ótima por informações calculadas — para os exercícios.

A estrutura de um caminho mínimo

Começaremos caracterizando a estrutura de uma solução ótima. Para o problema de caminhos mínimos para todos os pares em um grafo $G = (V, E)$, já provamos (Lema 24.1) que todos os subcaminhos de um caminho mínimo são caminhos mínimos. Suponha que representemos o grafo por uma matriz de adjacências $W = (w_{ij})$. Considere um caminho mínimo p do vértice i ao vértice j, e suponha que p contenha no máximo m arestas. Considerando que não existe nenhum ciclo de peso negativo, m é finito. Se $i = j$, então p tem peso 0 e nenhuma aresta. Se os vértices i e j são distintos, decompomos o caminho p em $i\, p'\, k \rightarrow j$, onde o caminho p' agora contém no máximo $m - 1$ arestas. Pelo Lema 24.1, p' é um caminho mínimo de i a k e, assim, $d(i, j) = d(i, k) + w_{kj}$.

Uma solução recursiva para o problema de caminhos mínimos para todos os pares

Agora, seja $l_{ij}^{(m)}$ o peso mínimo de qualquer caminho do vértice i ao vértice j, que contém no máximo m arestas. Quando $m = 0$, existe um caminho mínimo de i a j sem nenhuma aresta se e somente se $i = j$. Portanto,

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{se } i = j\ , \\ \infty & \text{se } i \neq j\ . \end{cases}$$

Para m 1, calculamos $l_{ij}^{(m)}$ como o mínimo de $l_{ik}^{(m-1)}$ (o peso de um caminho mínimo de i a j que consiste no máximo em $m - 1$ arestas) e o peso mínimo de qualquer caminho de i a j que consiste no máximo em m arestas, obtido pelo exame de todos os possíveis predecessores k de j. Assim, definimos recursivamente

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\le k\le n}\{l_{ik}^{(m-1)} + w_{kj}\}\right) \\ &= \min_{1\le k\le n}\{l_{ik}^{(m-1)} + w_{kj}\}\ . \end{aligned} \tag{25.2}$$

A última igualdade decorre de que $w_{jj} = 0$ para todo j.

Quais são os pesos reais de caminhos mínimos $d(i, j)$? Se o grafo não contém nenhum ciclo de peso negativo, então para todo par de vértices i e j para o qual $d(i, j) < \infty$, existe um caminho mínimo de i a j que é simples e por isso contém no máximo $n - 1$ arestas. Um caminho do vértice i ao vértice j com mais de $n - 1$ arestas não pode ter peso menor que um caminho mínimo de i a j. Então, os pesos reais de caminhos mínimos são dados por

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots\ . \tag{25.3}$$

Cálculo dos pesos de caminhos mínimos de baixo para cima

Tomando como nossa entrada a matriz $W = (w_{ij})$, calculamos agora uma série de matrizes $L^{(1)}, L^{(2)}, ..., L^{(n-1)}$, onde, para $m = 1, 2, ..., n - 1$, temos $L^{(m)} = \left(l_{ij}^{(m)}\right)$. A matriz final $L^{(n-1)}$ contém os pesos reais de caminhos mínimos. Observe que $l_{ij}^{(1)} = w_{ij}$ para todos os vértices $i, j \in V$ e, assim, $L^{(1)} = W$.

O núcleo do algoritmo é o procedimento a seguir que, dadas as matrizes $L^{(m-1)}$ e W, retorna a matriz $L^{(m)}$. Isto é, ele estende por mais uma aresta os caminhos mínimos calculados até agora.

EXTEND-SHORTEST-PATHS(L, W)

```
1   n = L.linhas
2   seja L′ = uma nova matriz n × n (l'_ij).
3   for i = 1 to n
4      for j = 1 to n
5          l'_ij = ∞
6          for k = 1 to n
7      l'_ij = min(l'_ij , l_ik + w_kj)
8   return L′
```

O procedimento calcula uma matriz $L' = (l'_{ij})$, que ele devolve no final. Para tal, calcula a equação (25.2) para todo i e j, utilizando L para $L^{(m-1)}$ e L' para $L^{(m)}$. (O algoritmo é escrito sem os índices para que suas matrizes de entrada e saída sejam independentes de m.) Seu tempo de execução é $Q(n_3)$, devido aos três laços **for** aninhados.

Agora podemos ver a relação com a multiplicação de matrizes. Suponha que desejemos calcular o produto de matrizes $C = A \cdot B$ de duas matrizes $n \times n$ A e B. Então, para $i, j = 1, 2, ..., n$, calculamos

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}. \tag{25.4}$$

Observe que, se fizermos as substituições

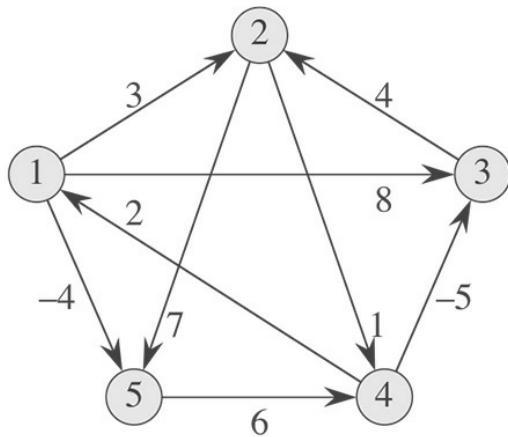
$$\begin{aligned} l^{(m-1)} &\to a\,, \\ w &\to b\,, \\ l^{(m)} &\to c\,, \\ \min &\to +\,, \\ + &\to \cdot \end{aligned}$$

na equação (25.2), obtemos a equação (25.4). Assim, se fizermos essas mudanças em EXTEND-SHORTEST-PATHS e também substituirmos ∞ (a identidade para *min*) por 0 (a identidade para +), obtemos o mesmo procedimento de tempo $Q(n_3)$ para multiplicar matrizes quadradas que vimos na Seção 4.2:

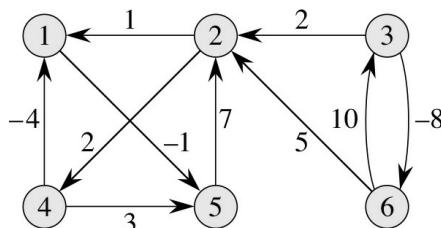
```
SQUARE-MATRIX-MULTIPLY (A,B)

1   n = A.linhas
2   seja C uma nova matriz n × n
3   for i = 1 to n
4       for j = 1 to n
5           c_ij = 0
6           for k = 1 to n
7               c_ij = c_ij + a_ik · b_kj
8   return C
```

Retornando ao problema de caminhos mínimos para todos os pares, calculamos os pesos de caminhos mínimos estendendo os caminhos mínimos aresta por aresta. Denotando por $A \cdot B$ o "produto" de matrizes retornado por EXTEND-SHORTEST-PATHS(A, B), calculamos a sequência de $n - 1$ matrizes

$$\begin{aligned} L^{(1)} &= L^{(0)} . W &= W , \\ L^{(2)} &= L^{(1)} . W &= W^2 , \\ L^{(3)} &= L^{(2)} . W &= W^3 , \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W &= W^{n-1}. \end{aligned}$$

Como demonstramos, a matriz $L_{(n} - {}^{1)} = W_n - {}^1$ contém os pesos de caminhos mínimos. O procedimento a seguir, calcula essa sequência no tempo Q(n_4).

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1   n = W.linhas
2   L^(1) = W
3   for m = 2 to n − 1
4       seja L^(m) uma nova matriz n × n
5       L^(m) = EXTEND-SHORTEST-PATHS(L^(m − 1), W)
6   return L^(n − 1)
```

A Figura 25.1 mostra um grafo e as matrizes $L_{(}{}^{m})$ calculadas pelo procedimento SLOW-ALL- PAIRS-SHORTEST-PATHS.

Melhorar o tempo de execução

Nosso objetivo, entretanto, não é calcular *todas* as matrizes $L_{(}{}^{m})$: estamos interessados somente na matriz $L_{(n} - {}^{1)}$. Lembre-se de que, na ausência de ciclos de peso negativo, a equação (25.3) implica $L_{(}{}^{m}) = L_{(n} - {}^{1)}$ para todos os inteiros $m \geq n - 1$. Exatamente como a multiplicação de matrizes tradicional é associativa, também é associativa a multiplicação de matrizes definida pelo procedimento EXTEND-SHORTEST-PATHS (veja o Exercício 25.1-4). Portanto, podemos calcular $L_{(n} - {}^{1)}$ com somente lg($n - 1$) produtos de matrizes calculando a sequência

$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Figura 25.1 Um grafo dirigido e a sequência de matrizes $L^{(m)}$ calculada por SLOW-ALL-PAIRS-SHORTEST-PATHS. O leitor pode verificar que $L^{(5)}$, definido como $L^{(4)} \cdot W$ é igual a $L^{(4)}$ e, assim, $L^{(m)} = L^{(4)}$ para todo $m \geq 4$.

Figura 25.2 Um grafo dirigido ponderado para uso nos Exercícios 25.1-1, 25.2-1 e 25.3-1.

$$L^{(1)} = W$$
$$L^{(2)} = W^2 = W \,.\, W \,,$$
$$L^{(4)} = W^4 = W^2 \,.\, W^2$$
$$L^{(8)} = W^8 = W^4 \,.\, W^4 \,,$$
$$\vdots$$
$$L^{(2^{\lceil \lg(n-1) \rceil})} = W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot L^{2^{\lceil \lg(n-1) \rceil - 1}}$$

Visto que $2^{\lg(n-1)} \geq n - 1$, o produto final $L_{(2^{\lg(n-1)})}$ é igual a $L_{(n-1)}$.

O procedimento a seguir calcula a mesma sequência de matrizes empregando essa técnica de ***elevação ao quadrado repetida***.

FASTER-ALL-PAIRS-SHORTEST-PATHS(W)

```
1   n = W.linhas
2   L(1) = W
3   m = 1
4   while m < n − 1
5       seja L(2m) uma nova matriz n × n
6       L(2m) = EXTEND-SHORTEST-PATHS(L(m), L(m))
7       m = 2m
8   return L(m)
```

Em cada iteração do laço **while** das linhas 4–7, calculamos $L_{(2^m)} = (L_{(^m)})^2$, começando com $m = 1$. No final de cada iteração, dobramos o valor de m. A iteração final calcula $L_{(n-1)}$ calculando na realidade $L_{(2^m)}$ para algum $n - 1 \leq 2m \leq 2n - 2$. Pela equação (25.3), $L_{(2^m)} = L_{(n-1)}$. Na próxima vez que o teste da linha 4 for executado, m já estará com seu valor dobrado; portanto, agora, $m \geq n - 1$, o teste falha e o procedimento devolve a última matriz que calculou.

Como cada um dos produtos de matrizes demora o tempo $Q(n_3)$, FASTER-ALL-PAIRS-SHOR- TEST-PATHS é executado no tempo $Q(n_3 \lg n)$. Observe que o código é compacto, não contém nenhuma estrutura de dados elaborada e, portanto, a constante oculta na notação Q é pequena.

Exercícios

25.1-1 Execute SLOW-ALL-PAIRS-SHORTEST-PATHS no grafo dirigido ponderado da Figura 25.2, mostrando as matrizes que resultam para cada iteração do laço. Depois, faça o mesmo para FASTER-ALL-PAIRS-SHORTEST-PATHS.

25.1-2 Por que exigimos que $w_{ii} = 0$ para todo $1 \leq i \leq n$?

25.1-3 A que corresponde, na multiplicação de matrizes comum, a matriz

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \dots & \infty \\ \infty & 0 & \infty & \dots & \infty \\ \infty & \infty & 0 & \dots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \dots & 0 \end{pmatrix}$$

usada nos algoritmos?

25.1-4 Mostre que a multiplicação de matrizes definida por EXTEND-SHORTEST-PATHS é associativa.

25.1-5 Mostre como expressar o problema de caminhos mínimos de fonte única como um produto de matrizes e um vetor. Descreva como a avaliação desse produto corresponde a um algoritmo como o de Bellman-Ford (veja a Seção 24.1).

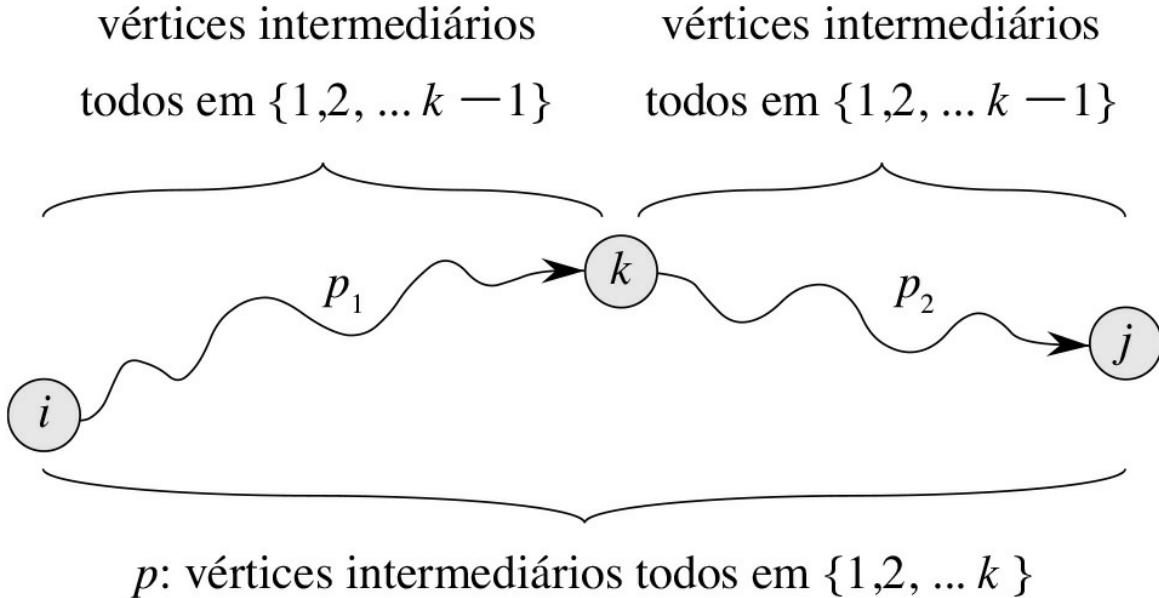
25.1-6 Suponha que também desejemos calcular os vértices em caminhos mínimos nos algoritmos desta seção. Mostre como calcular a matriz de predecessores P pela matriz completada L de pesos de caminhos mínimos no tempo $O(n_3)$.

25.1-7 Podemos calcular também os vértices em caminhos mínimos à medida que calculamos os pesos de caminhos mínimos. Defina $(m)_{ij}$ como o predecessor do vértice j em qualquer caminho de peso mínimo de i a j que contém no máximo m arestas. Modifique os procedimentos EXTEND-SHORTEST-PATHS e SLOW-ALL-PAIRS-SHORTEST-PATHS para calcular as matrizes P(1), P(2), ..., P($_n$ - 1) à medida que as matrizes $L_{(1)}$, $L_{(2)}$, ..., $L_{(n}$ -1) são calculadas.

25.1-8 O procedimento FASTER-ALL-PAIRS-SHORTEST-PATHS, como foi escrito, requer o armazenamento de lg(n – 1) matrizes, cada uma com n_2 elementos, para um requisito de espaço total de Q(n_2 lg n). Modifique o procedimento para que o requisito de espaço seja somente Q(n_2), usando somente duas matrizes $n \times n$.

25.1-9 Modifique FASTER-ALL-PAIRS-SHORTEST-PATHS de modo que ele possa detectar se o grafo contém um ciclo de peso negativo.

25.1-10 Dê um algoritmo eficiente para encontrar o comprimento (número de arestas) de um ciclo de peso negativo de comprimento mínimo em um grafo.

25.2 O ALGORITMO DE FLOYD-WARSHALL

Nesta seção, usaremos uma formulação de programação dinâmica diferente para resolver o problema de caminhos mínimos para todos os pares em um grafo dirigido $G = (V, E)$. O algoritmo resultante, conhecido como ***algoritmo de Floyd-Warshall***, é executado no tempo Q(V_3). Como antes, arestas de peso negativo podem estar presentes, mas supomos que não existe nenhum ciclo de peso negativo. Como na Seção 25.1, seguiremos o processo de programação dinâmica para desenvolver o algoritmo. Depois de estudar o algoritmo resultante, apresentamos um método semelhante para encontrar o fecho transitivo de um grafo dirigido.

A estrutura de um caminho mínimo

No algoritmo de Floyd-Warshall, caracterizamos a estrutura de um caminho mínimo de um modo diferente do que usamos na Seção 25.1. O algoritmo de Floyd-Warshall considera os vértices intermediários de um caminho mínimo,

onde um vértice ***intermediário*** de um caminho simples $p = \langle v_1, v_2, ..., v_l \rangle$ é qualquer vértice de p exceto v_1 ou v_l, isto é, qualquer vértice no conjunto $\{v_2, v_3, ..., v_{l-1}\}$.

O algoritmo de Floyd-Warshall se baseia na seguinte observação: como supomos que os vértices de G são $V = \{1, 2, ..., n\}$, vamos considerar um subconjunto $\{1, 2, ..., k\}$ de vértices para algum k. Para qualquer par de vértices $i, j \in V$, considere todos os caminhos de i a j cujos vértices intermediários estejam em $\{1, 2, ..., k\}$, e seja p um caminho de peso mínimo dentre eles. (O caminho p é simples.) O algoritmo de Floyd-Warshall explora uma relação entre o caminho p e os caminhos mínimos de i a j cujos vértices intermediários estejam todos no conjunto $\{1, 2, ..., k - 1\}$. A relação depende de k ser ou não um vértice intermediário do caminho p.

Figura 25.3 O caminho p é um caminho mínimo do vértice i ao vértice j e k é o vértice intermediário de p com numeração mais alta. O caminho p_1, a porção de caminho p do vértice i ao vértice k, tem todos os vértices intermediários no conjunto $\{1, 2, ..., k - 1\}$. O mesmo vale para o caminho p_2 do vértice k ao vértice j.

- Se k não é um vértice intermediário do caminho p, então todos os vértices intermediários do caminho p estão no conjunto $\{1, 2, ..., k - 1\}$. Assim, um caminho mínimo do vértice i ao vértice j cujos vértices intermediários estão no conjunto $\{1, 2, ..., k - 1\}$ também é um caminho mínimo de i a j cujos vértices intermediários estão no conjunto $\{1, 2, ..., k\}$.
- Se k é um vértice intermediário do caminho p, então desmembramos p em $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$, como mostra a Figura 25.3. Pelo Lema 24.1, p_1 é um caminho mínimo de i a k cujos vértices intermediários estão no conjunto $\{1, 2, ..., k\}$. Na verdade, a nossa afirmativa pode ser um pouco mais contundente. Como o vértice k não é um vértice intermediário do caminho p_1, todos os vértices intermediários de p_1 estão no conjunto $\{1, 2, ..., k - 1\}$. Portanto, p_1 é um caminho mínimo de i a k cujos vértices intermediários estão no conjunto $\{1, 2, ..., k - 1\}$. De modo semelhante, p_2 é um caminho mínimo do vértice k ao vértice j cujos vértices intermediários estão no conjunto $\{1, 2, ..., k - 1\}$.

Uma solução recursiva para o problema de caminhos mínimos para todos os pares

Com base nas observações anteriores, definimos uma formulação recursiva de estimativas de caminhos mínimos diferente da que definimos na Seção 25.1. Seja $d_{ij}^{(k)}$ o peso de um caminho mínimo do vértice i ao vértice j cujos vértices intermediários estão no conjunto $\{1, 2, ..., k\}$. Quando $k = 0$, um caminho do vértice i ao vértice j que não tem nenhum vértice intermediário com numeração mais alta que 0 não tem absolutamente nenhum vértice intermediário. Tal caminho tem, no máximo, uma aresta e então $d^{(0)} = w$. Conforme essa discussão, definimos $d_{ij}^{(k)}$ recursivamente por

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{se } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{se } k \geq 1. \end{cases} \quad (25.5)$$

Considerando que, para qualquer caminho, todos os vértices intermediários estão no conjunto {1, 2, ..., n}, a matriz $D_{(n)}) = (d(n)_{ij}$) dá a resposta final: $d(n)_{ij} = d(i, j)$ para todo $i, j \in V$.

Calculando os pesos de caminhos mínimos de baixo para cima

Com base na recorrência (25.5), podemos usar o procedimento de baixo para cima dado a seguir para calcular os valores $d(k)_{ij}$em ordem crescente de valores de k. Sua entrada é uma matriz $n \times n$ W definida como na equação (25.1). O procedimento devolve a matriz $D_{(n)}$) de pesos de caminhos mínimos.

FLOYD-WARSHALL(W)

```
1  n = W.linhas
2  D^(0) = W
3  for k = 1 to n
4      seja D^(k) = (d_ij^(k)) uma nova matriz n × n
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

A Figura 25.4 mostra as matrizes $D_{(}k$) calculadas pelo algoritmo de Floyd-Warshall para o grafo na Figura 25.1.

O tempo de execução do algoritmo de Floyd-Warshall é determinado pelos laços **for** triplamente aninhados das linhas 3–7. Como cada execução da linha 7 demora o tempo $O(1)$, o algoritmo é executado no tempo Q(n_3). Como no algoritmo final na Seção 25.1, o código é compacto, sem nenhuma estrutura de dados elaborada, portanto a constante oculta na notação Q é pequena. Assim, o algoritmo de Floyd-Warshall é bastante prático até mesmo para grafos de entrada de dimensões moderadas.

Como construir um caminho mínimo

Existe uma variedade de métodos diferentes para construir caminhos mínimos no algoritmo de Floyd-Warshall. Um deles é calcular a matriz D de pesos de caminhos mínimos e depois construir a matriz de predecessores P pela matriz D. O Exercício 25.1-6 pede que você implemente esse método de modo que seja executado no tempo $O(n_3)$. Dada a matriz de predecessores P, o procedimento PRINT-ALL-PAIRS-SHORTEST-PATH imprimirá os vértices em um caminho mínimo dado.

Alternativamente, podemos calcular a matriz de predecessores P enquanto o algoritmo calcula as matrizes $D_{(}k$). Especificamente, calculamos uma sequência de matrizes P(0), P(1), ..., P($_n$), onde P = P($_n$), e definimos (k)$_{ij}$ como o predecessor do vértice j em um caminho mínimo que parte do vértice i cujos vértices intermediários estão no conjunto {1, 2, ..., k}.

Podemos dar uma formulação recursiva de (k)$_{ij}$. Quando k = 0, um caminho mínimo de i a j não tem absolutamente nenhum vértice intermediário. Assim,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ ou } w_{ij} = \infty , \\ i & \text{se } i \neq j \text{ e } w_{ij} < \infty . \end{cases} \quad (25.6)$$

Para $k \geq 1$, se tomarmos o caminho $i \leadsto k \leadsto j$, onde $k \neq j$, então o predecessor de j que escolhemos será igual ao predecessor de j que escolhemos em um caminho mínimo que parte de k cujos vértices intermediários estão no conjunto $\{1, 2, ..., k - 1\}$. Caso contrário, escolhemos o mesmo predecessor de j que escolhemos em um caminho mínimo que parte de i cujos vértices intermediários estão no conjunto $\{1, 2, ..., k - 1\}$. Formalmente, para $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \qquad (25.7)$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figura 25.4 A sequência de matrizes $D_{(k)}$ e $P(k)$ calculada pelo algoritmo de Floyd-Wharshall para o grafo na Figura 25.1.

Deixamos a incorporação dos cálculos das matrizes $P^{(k)}$ ao procedimento de FLOYD-WAR-SHALL para o Exercício 25.2-3. A Figura 25.4 mostra a sequência de matrizes $P^{(k)}$ que o algoritmo resultante calcula para o grafo da Figura 25.1. O exercício também pede que você realize a tarefa mais difícil de provar que o subgrafo dos predecessores $G_{p,i}$ é uma árvore de caminhos mínimos com raiz i. O Exercício 25.2-7 pede ainda uma outra maneira de reconstruir caminhos mínimos.

Fecho transitivo de um grafo dirigido

Dado um grafo dirigido $G = (V, E)$ com o conjunto de vértices $V = \{1, 2, ..., n\}$, desejamos determinar se G contém um caminho de i a j para todos os pares de vértices $i, j \in V$. Definimos o ***fecho transitivo*** de G como o grafo $G^* = (V, E^*)$, onde

$$E^* = \{(i, j) : \text{existe um caminho do vértice } i \text{ ao vértice } j \text{ em } G\}\ .$$

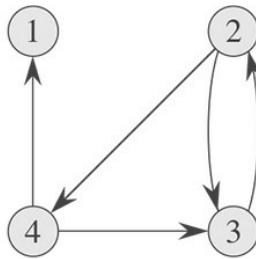
Um modo de calcular o fecho transitivo de um grafo no tempo $Q(n_3)$ é atribuir peso 1 a cada aresta de E e executar o algoritmo de Floyd-Warshall. Se existe um caminho do vértice i ao vértice j, obtemos $d_{ij} < n$. Caso contrário, obtemos $d_{ij} = \infty$.

Há um outro modo semelhante de calcular o fecho transitivo de G no tempo $O(n_3)$ que pode poupar tempo e espaço na prática. Esse método substitui as operações aritméticas min e + no algoritmo de Floyd-Warshall pelas operações lógicas $\vee$ (OU lógico) e $\wedge$ (E lógico). Para $i, j, k = 1, 2, ..., n$, definimos $t^{(k)}_{ij}$. como 1 se existe um caminho no grafo G do vértice i ao vértice j com todos os vértices intermediários no conjunto $\{1, 2, ..., k\}$ e como 0 em caso contrário. Construímos o fecho transitivo $G^* = (V E^*)$ inserindo a aresta $(i\ j)$ em E^* se e somente se $t^{(n)}_{ij} = 1$. Uma definição recursiva de $t^{(k)}$ análoga à recorrência (25.5) é

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i,j) \notin E\ , \\ 1 & \text{se } i = j \text{ ou} (i,j) \in E\ , \end{cases}$$

e para $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right). \tag{25.8}$$

Como no algoritmo de Floyd-Warshall, calculamos as matrizes $T^{(k)} = \left(t_{ij}^{(k)}\right)$ em ordem crescente de k.

TRANSITIVE-CLOSURE (G)

1 $n = |G.V|$
2 seja $T^{(0)} = \left(t_{ij}^{(0)}\right)$ uma nova matriz $n \times n$
3 **for** $i = 1$ **to** n
4   **for** $j = 1$ **to** n
5   **if** $i == j$ ou $(i, j) \in G.E$
6             $t_{ij}^{(0)} = 1$
7   **else** $t_{ij}^{(0)} = 0$
8 **for** $k = 1$ **to** n
9   seja $T^{(k)} = \left(t_{ij}^{(k)}\right)$ uma nova matriz $n \times n$
10   **for** $i = 1$ **to** n
11   **for** $j = 1$ **to** n
12                 $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
13   **return** $T^{(n)}$

A Figura 25.5 mostra as matrizes $T^{(k)}$ calculadas pelo procedimento TRANSITIVE-CLOSURE em um grafo de amostra. O procedimento TRANSITIVE-CLOSURE, assim como o algoritmo de Floyd-Warshall, é executado no tempo $Q(n_3)$. Entretanto, em alguns computadores, operações lógicas em valores de um único bit são executadas mais rapidamente que operações aritméticas em palavras de dados inteiras. Além disso, como o algoritmo direto de fecho transitivo usa somente valores booleanos em vez de valores inteiros, seu requisito de espaço é menor que o do algoritmo de FLOYD-WARSHALL por um fator correspondente ao tamanho de uma palavra de armazenamento no computador.

$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figura 25.5 Um grafo dirigido e as matrizes $T^{(k)}$ calculadas pelo algoritmo de fecho transitivo.

Exercícios

25.2-1 Execute o algoritmo de Floyd-Warshall no grafo dirigido ponderado da Figura 25.2. Mostre a matriz $D_{(}^{k)}$ que resulta de cada iteração do laço externo.

25.2-2 Mostre como calcular o fecho transitivo empregando a técnica da Seção 25.1.

25.2-3 Modifique o procedimento FLOYD-WARSHALL para calcular as matrizes P(k) de acordo com as equações (25.6) e (25.7). Prove rigorosamente que, para todo $i \in V$, o subgrafo dos predecessores G_{P}, ^{i} é uma árvore de caminhos mínimos com raiz i. (*Sugestão*: Para mostrar que G_{P}, ^{i} é acíclico, primeiro mostre que $(k)_{ij} = l$ implica $d(k)_{ij} \geq d\ (k)_{ij} + w\ _{ij}$, de acordo com a definição de $(k)_{ij}$. Então adapte a prova do Lema 24.16.)

25.2-4 Como foi apresentado, o algoritmo de Floyd-Warshall requer o espaço Q(n_3), visto que calculamos $d(_{k}$) para i, j, k = 1, 2, ..., n. Mostre que o procedimento a seguir, que simplesmente descarta todos os índices superiores, é correto e, assim, o espaço requerido é somente Q(n_2).

```
FLOYD-WARSHALL'(W)
1 n = W.linhas
2 D = W
3 for k = 1 to n
4         for i = 1 to n
5                 for j = 1 to n
6                         d_ij = min(d_ij, d_ik + d_kj)
7 return D
```

25.2-5 Suponha que modificamos o modo como a equação (25.7) trataa igualdade:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k)} & \text{se } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Essa definição alternativa da matriz de predecessores P é correta?

25.2-6 Como podemos usar a saída do algoritmo de Floyd-Warshall para detectar a presença de um ciclo de peso negativo?

25.2-7 Um outro modo de reconstruir caminhos mínimos no algoritmo de FLOYD-WARSHALL utiliza valores $(k)_{ij}$ para i, j, k = 1, 2, ..., n, onde $(k)_{ij}$ é o vértice intermediário de numeração mais alta de um caminho mínimo de i a j no qual todos os vértices intermediários estão no conjunto {1, 2, ..., k}. Apresente uma formulação recursiva para $(k)_{ij}$, modifique o procedimento FLOYD-WARSHALL para calcular os valores de $(k)_{ij}$ e reescreva o procedimento PRINT-ALL-PAIRS-SHORTEST-PATH para adotar a matriz $= \left(\phi_{ij}^{(n)}\right)$ como entrada. Qual é a semelhança entre a matriz e a tabela s no problema de multiplicação de cadeias de matrizes da Seção 15.2?

25.2-8 Dê um algoritmo de tempo $O(V\,E)$ para calcular o fecho transitivo de um grafo dirigido $G = (V, E)$.

25.2-9 Suponha que possamos calcular o fecho transitivo de um grafo acíclico no tempo $f(|V|, |E|)$, onde f é uma função monotonicamente crescente de $|V|$ e $|E|$. Mostre que o tempo para calcular o fecho transitivo $G_{*} = (V, E_{*})$ de um grafo dirigido geral $G = (V, E)$ é $f(|V|, |E|) + O(V + E^{*})$.

25.3 ALGORITMO DE JOHNSON PARA GRAFOS ESPARSOS

O algoritmo de Johnson encontra caminhos mínimos entre todos os pares no tempo $O(V_2 \lg V + VE)$. Para grafos esparsos, ele é assintoticamente melhor que a elevação ao quadrado repetida de matrizes ou o algoritmo de Floyd-Warshall. O algoritmo retorna uma matriz de pesos de caminhos mínimos para todos os pares de vértices ou informa que o grafo de entrada contém um ciclo de peso negativo. O algoritmo de Johnson usa como sub-rotinas o algoritmo de Dijkstra e o algoritmo de Bellman-Ford, descritos no Capítulo 24.

O algoritmo de Johnson emprega a técnica de ***reponderação***, que funciona da maneira descrita a seguir. Se todos os pesos de arestas w em um grafo $G = (V, E)$ são não negativos, podemos encontrar caminhos mínimos entre todos os pares de vértices executando o algoritmo de Dijkstra uma vez a partir de cada vértice; com a fila de prioridade mínima do heap de Fibonacci, o tempo de execução desse algoritmo para todos os pares é $O(V_2 \lg V + VE)$.

Se G tem arestas de peso negativo mas nenhum ciclo de peso negativo, simplesmente calculamos um novo conjunto de pesos de arestas não negativos que nos permita utilizar o mesmo método. O novo conjunto de pesos de arestas deve satisfazer duas propriedades importantes.

1. Para todos os pares de vértices $u, v \in V$, um caminho p é um caminho mínimo de u a v
usando a função peso w se e somente se p também é um caminho mínimo de u a v usando a função peso $\hat{w}$.
2. Para todas as arestas (u, v), o novo peso $\hat{w}(u, v)$ é não negativo.

Como veremos em breve, podemos pré-processar G para determinar a nova função peso no tempo $O(V E)$.

Preservando caminhos mínimos por reponderação

O lema a seguir mostra como é fácil reponderar as arestas para satisfazer a propriedade descrita no item 1. Utilizamos d para denotar pesos de caminhos mínimos derivados da função peso w e para denotar pesos de caminhos mínimos derivados da função peso $\hat{w}$.

Lema 25.1 (Reponderação não muda caminhos mínimos)

Dado um grafo dirigido ponderado $G = (V, E)$ com função peso $w : E \rightarrow$, seja $h : V \rightarrow$ qualquer função que mapeie vértices para números reais. Para cada aresta $(u, v) \in E$, defina

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v). \tag{25.9}$$

Seja $p = \langle v_0, v_1, ..., v_k \rangle$ qualquer caminho do vértice v_0 ao vértice v_k. Então, p é um caminho mínimo de v_0 a v_k com função peso w se e somente se é um caminho mínimo com função peso $\hat{w}$. Isto é, $w(p) = d(v_0, v_k)$ se e somente se $\hat{w}(p) = d(v_0, v_k)$. Além disso, G tem um ciclo de peso negativo usando função peso w se e somente se G tem um ciclo de peso negativo usando função peso $\hat{w}$.

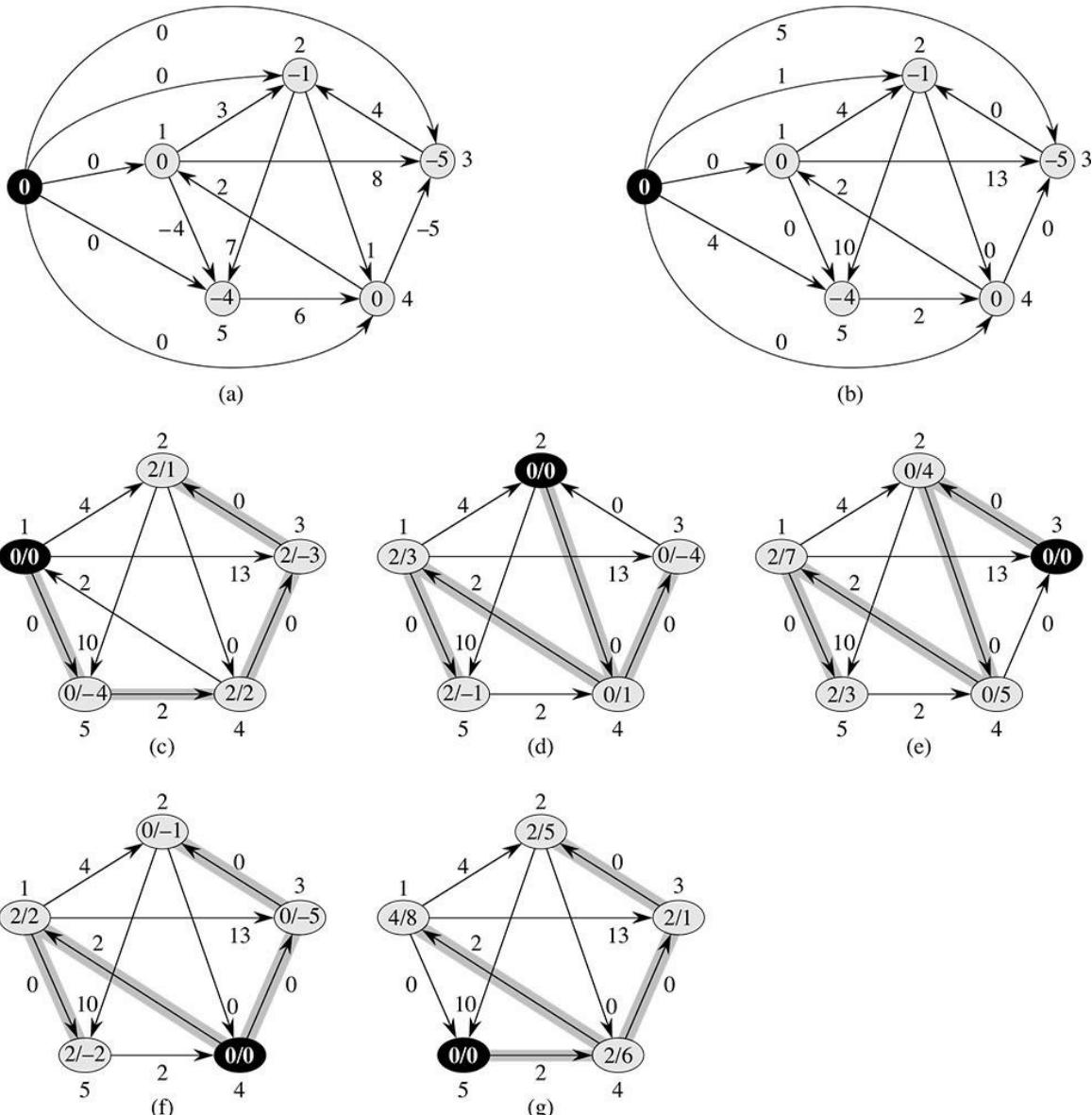
Prova Começamos mostrando que

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k) \tag{25.10}$$

Temos

$$\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \qquad \text{(porque a soma é telescópica)} \\
&= w(p) + h(v_0) - h(v_k).
\end{aligned}$$

Portanto, qualquer caminho p de v_0 a v_k tem $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$. Como $h(v_0)$ e $h(v_k)$ não dependem do caminho, se um caminho de v_0 a v_k é mais curto que outro usando função peso w, então, ele também é mais curto usando $\hat{w}$. Assim, $w(p) = d(v_0, v_k)$ se e somente se $\hat{w}(p) = \hat{}(v_0, v_k)$.
Finalmente, mostramos que G tem um ciclo de peso negativo usando a função peso w se e somente se G tem um ciclo de peso negativo usando a função peso $\hat{w}$. Considere qualquer ciclo $c = \langle v_0, v_1, ..., v_k \rangle$ onde $v_0 = v_k$. Pela equação (25.10),

$$\begin{aligned} \hat{w} &= w(c) + h(v_0) - h(v_k) \\ &= w(c), \end{aligned}$$

e, assim, c tem peso negativo usando w se e somente se tem peso negativo usando $\hat{w}$.

Produzindo pesos não negativos por reponderação

Nossa próxima meta é garantir que a segunda propriedade seja válida: queremos que $\hat{w}(u, v)$ seja não negativo para todas as arestas $(u, v) \in E$. Dado um grafo dirigido ponderado $G = (V, E)$ com função peso $w : E \rightarrow$, criamos um novo grafo $G' = (V', E')$, onde $V' = V \cup \{s\}$ para algum novo vértice $s \in V$ e $E' = E \cup \{(s, v) : v \in V\}$. Estendemos a função peso w de modo que $w(s, v) = 0$ para todo $v \in V$. Observe que, por s não ter nenhuma aresta de entrada, nenhum caminho mínimo em G', exceto os que partem de s, contém s. Além disso, G' não tem nenhum ciclo de peso negativo se e somente se G não tem nenhum ciclo de peso negativo. A Figura 25.6(a) mostra o grafo G' correspondente ao grafo G da Figura 25.1.

Agora, suponha que G e G' não tenham nenhum ciclo de peso negativo. Vamos definir $h(v) = d(s, v)$ para todo $v \in V'$. Pela desigualdade triangular (Lema 24.10), temos $h(v) \leq h(u) + w(u, v)$ para todas as arestas $(u, v) \in E'$. Portanto, se definirmos os novos pesos de acordo com a equação (25.9), teremos $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$, e satisfazemos a segunda propriedade. A Figura 25.6(b) mostra o grafo G' da Figura 25.6(a) com arestas reponderadas.

Figura 25.6 O algoritmo de caminhos mínimos para todos os pares de Johnson executado no grafo da Figura 25.1. A numeração dos vértices aparece fora deles. (**a**) O grafo *G'* com a função peso original *w*. O novo vértice *s* é preto. Dentro de cada vértice *v* está $h(v) = d(s, v)$**.** (**b**) Após a reponderação de cada aresta (u, v) com a função peso $\hat{w}(u, v) = w(u, v) + h(v)$. (**c**)-(**g**) Resultado da execução do algoritmo de Dijkstra em cada vértice de *G* usando a função peso $\hat{w}$. Em cada parte, o vértice de fonte *u* é preto, e as arestas sombreadas estão na árvore de caminhos mínimos calculada pelo algoritmo. Dentro de cada vértice *v* estão os valores ˆ(u, v) e $d(u, v)$, separados por uma barra inclinada. O valor $d_{uv} = d(u, v)$ é igual a ˆ$(u, v) + h(v) - h(u)$.

Calculando caminhos mínimos para todos os pares

O algoritmo de Johnson para calcular caminhos mínimos para todos os pares emprega o algoritmo de Bellman-Ford (Seção 24.1) e o algoritmo de Dijkstra (Seção 24.3) como sub-rotinas, e supõe implicitamente que as arestas estão armazenadas em listas de adjacências. O algoritmo retorna a matriz $|V| \times |V|$ habitual $D = d_{ij}$, onde $d_{ij} = d(i, j)$ ou informa que o grafo de entrada contém um ciclo de peso negativo. Como é típico, no caso de um algoritmo de caminhos mínimos para todos os pares, supomos que os vértices são numerados de 1 a $|V|$.

JOHNSON(G)

```
1   calcular G′, onde V[G′] = V[G] ∪ {s},
        E[G′] = E[G] ∪ {(s, v) : v ∈ V[G]} e
        w(s, v) = 0 para todo v ∈ V[G]
2   if BELLMAN-FORD(G′, w, s) == FALSE
3       imprimir "o grafo de entrada contém um ciclo de peso negativo"
4   else for cada vértice v ∈ V[G′]
5       definir h(v) como o valor de δ(s, v)
            calculado pelo algoritmo de Bellman-Ford
6   for cada aresta (u, v) ∈ E[G′]
7       ŵ(u, v) = w(u, v) + h(u) − h(v)
8   seja D = (d_uv) uma nova matriz n × n
9   for cada vértice u ∈ V[G]
10      executar DIJKSTRA(G, ŵ, u) para calcular δ̂(u, v) para todo v ∈ V[G]
11      for cada vértice v ∈ V[G]
12          d_uv = δ̂(u, v) + h(v) − h(u)
13  return D
```

Esse código simplesmente executa as ações que especificamos anteriormente. A linha 1 produz G'. A linha 2 executa o algoritmo de Bellman-Ford em G' com função peso w e vértice fonte s. Se G', e consequentemente G, contém um ciclo de peso negativo, a linha 3 relata o problema. As linhas 4–12 consideram que G' não contém nenhum ciclo de peso negativo. As linhas 4–5 definem $h(v)$ como o peso do caminho mínimo $d(s, v)$ calculado pelo algoritmo de Bellman-Ford para todo $v \in V'$. As linhas 6–7 calculam os novos pesos $w\hat{}$. Para cada par de vértices $u, v \in V$, o laço **for** das linhas 9–11 calcula o peso do caminho mínimo $\hat{}(u, v)$ chamando o algoritmo de Dijkstra uma vez para cada vértice em V. A linha 12 armazena na entrada de matriz d_{uv} o peso correto do caminho mínimo $d(u, v)$, calculado pela equação (25.10). Finalmente, a linha 13 retorna a matriz D completada. A Figura 25.6 mostra a execução do algoritmo de Johnson.

Se implementarmos a fila de prioridade mínima no algoritmo de Dijkstra por um heap de Fibonacci, o algoritmo de Johnson é executado no tempo $O(V_2 \lg V + VE)$. A implementação mais simples por heap mínimo binário produz o tempo de execução $O(VE \lg V)$, que ainda é assintoticamente mais rápido que o algoritmo de Floyd-Warshall se o grafo é esparso.

Exercícios

25.3-1 Use o algoritmo de Johnson para encontrar os caminhos mínimos entre todos os pares de vértices no grafo da Figura 25.2. Mostre os valores de h e $w\hat{}$ calculados pelo algoritmo.

25.3-2 Qual é a finalidade de adicionar o novo vértice s a V, produzindo V'?

25.3-3 Suponha que $w(u, v) \geq 0$ para todas as arestas $(u, v) \in E$. Qual é a relação entre as funções peso w e $w\hat{}$?

25.3-4 O professor Greenstreet afirma que existe um modo mais simples de reponderar arestas que o método usado no algoritmo de Johnson. Fazendo $w^* = \min_{(u, v) \in E} \{w(u, v)\}$, basta definir $w\hat{}\ (u, v) = w(u, v) - w^*$ para todas as arestas $(u, v) \in E$. O que está errado no método de reponderação do professor?

25.3-5 Suponha que executemos o algoritmo de Johnson em um grafo dirigido G com função peso w. Mostre que, se G contém um ciclo c de peso 0, então $w\hat{}\ (u, v) = 0$ para toda aresta (u, v) em c.

25.3-6 O professor Michener afirma que não há necessidade de criar um novo vértice de fonte na linha 1 de JOHNSON. Diz ele que, em vez disso, podemos simplesmente usar $G' = G$ e fazer s ser qualquer vértice. Dê um exemplo de grafo dirigido ponderado G para o qual a incorporação da ideia do professor em JOHNSON provoca respostas incorretas. Depois mostre que, se G é fortemente conexo (todo vértice pode ser alcançado de qualquer outro vértice), os resultados retornados por JOHNSON com a modificação do professor são corretos.

Problemas

25-1 ***Fecho transitivo de um grafo dinâmico***

Suponha que desejemos manter o fecho transitivo de um grafo dirigido $G = (V, E)$ à medida que inserimos arestas em E. Isto é, após a inserção de cada aresta, queremos atualizar o fecho transitivo das arestas inseridas até então. Suponha que, inicialmente, o grafo G não tenha nenhuma aresta e que representamos o fecho transitivo como uma matriz booleana.

a. Mostre como atualizar o fecho transitivo $G^* = (V, E^*)$ de um grafo $G = (V, E)$ no tempo $O(V_2)$ quando uma nova aresta é adicionada a G.

b. Dê um exemplo de grafo G e uma aresta e tal que seja necessário o tempo (V_2) para atualizar o fecho transitivo após a inserção de e em G, não importando qual algoritmo seja usado.

c. Descreva um algoritmo eficiente para atualizar o fecho transitivo à medida que arestas são inseridas no grafo. Para qualquer sequência de n inserções, seu algoritmo deve ser executado no tempo total $\sum_{i=1}^{n} t_i = O(V^3)$, onde t é o tempo para atualizar o fecho transitivo quando a i-ésima aresta é inserida. Prove que seu algoritmo consegue esse limite de tempo.

25-2 ***Caminhos mínimos em grafos e-densos***

Um grafo $G = (V, E)$ é ***-denso*** se $|E| = Q(V_1+)$ para alguma constante e na faixa $0 < e \leq 1$. Utilizando heaps de mínimo d-ários (veja o Problema 6-2) em algoritmos de caminhos mínimos em grafos e-densos, podemos alcançar os tempos de execução de algoritmos baseados em heaps de Fibonacci sem utilizar uma estrutura de dados tão complicada.

a. Quais são os tempos de execução assintóticos para INSERT, EXTRACT-MIN e DECREASE-KEY, em função de d e do número n de elementos em um heap d-ário? Quais são esses tempos de execução se escolhemos $d = Q(n_a)$ para alguma constante $0 < \alpha \leq 1$? Compare esses tempos de execução com os custos amortizados dessas operações para um heap de Fibonacci.

b. Mostre como calcular caminhos mínimos de fonte única em um grafo dirigido e-denso $G = (V, E)$ que não tenha nenhuma aresta de peso negativo no tempo $O(E)$. (*Sugestão*: Escolha d em função de e.)

c. Mostre como resolver o problema de caminhos mínimos para todos os pares em um grafo dirigido e-denso $G = (V, E)$ que não tenha nenhuma aresta de peso negativo no tempo $O(V E)$.

d. Mostre como resolver o problema de caminhos mínimos para todos os pares no tempo $O(V E)$ em um grafo dirigido e-denso $G = (V, E)$ que pode ter arestas de peso negativo, mas não tem nenhum ciclo de peso negativo.

NOTAS DO CAPÍTULO

Lawler [224] dá uma boa descrição do problema de caminhos mínimos para todos os pares, embora não analise soluções para grafos esparsos. Ele atribui o algoritmo de multiplicação de matrizes ao folclore. O algoritmo de Floyd-Warshall foi criado por Floyd [105], que tomou como base um teorema de Warshall [349] que descreve como calcular o fecho transitivo de matrizes booleanas. O algoritmo de Johnson foi obtido de [192].

Vários pesquisadores apresentaram algoritmos melhorados para calcular caminhos mínimos por multiplicação de matrizes. Fredman [111] mostra como resolver o problema de caminhos mínimos para todos os pares usando $(V_{5/2})$ comparações entre somas de pesos de arestas e obtém um algoritmo que é executado no tempo $O(V_3 (\lg \lg V / \lg V)^{1/3})$, ligeiramente melhor que o tempo de execução do algoritmo de Floyd-Warshall. Han [159] reduziu o tempo de execução para $O(V^3(\lg \lg V / \lg V)^{5/4})$. Outra linha de pesquisa demonstra que podemos aplicar algoritmos para multiplicação rápida de matrizes (veja as Notas do Capítulo 4) ao problema de caminhos mínimos para todos os pares. Seja $O(n_w)$ o tempo de execução do algoritmo mais rápido para multiplicar matrizes $n \times n$; atualmente, $w < 2{,}376$ [78]. Galil e Margalit [123, 124] e Seidel [308] criaram algoritmos que resolvem o problema de caminhos mínimos para todos os pares em grafos não dirigidos e não ponderados no tempo $(V_w\, p(V))$, onde $p(n)$ denota uma função específica que é limitada polilogaritmicamente em n. Em grafos densos, esses algoritmos são mais rápidos que o tempo $O(VE)$ necessário para executar $|V|$ pesquisas em largura. Vários pesquisadores estenderam esses resultados para dar algoritmos que resolvem o problema de caminhos mínimos para todos os pares em grafos não dirigidos nos quais os pesos de arestas são inteiros no intervalo $\{1, 2, ..., W\}$. Desses algoritmos, o mais rápido assintoticamente, criado por Shoshan e Zwick [316], é executado no tempo $O(W\, V_w\, p(VW))$.

Karger, Koller e Phillips [196] e McGeoch [215], independentemente, deram um limite de tempo que depende de E_*, o conjunto de arestas em E que participam de algum caminho mínimo. Dado um grafo com pesos de arestas não negativos, tais algoritmos são executados no tempo $O(VE_* + V_2 \lg V)$ e melhoram com a execução do algoritmo de Dijkstra $|V|$ vezes quando $|E_*| = o(E)$.

Baswana, Hariharan e Sen [33] examinaram algoritmos decrementadores para manter informações de caminhos mínimos para todos os pares e de fecho transitivo. Algoritmos decrementadores permitem uma sequência de eliminações e consultas entremeadas; por comparação, o Problema 25-1, no qual são inseridas arestas, pede um algoritmo incremental. Os algoritmos criados por Baswana, Hariharan e Sen são aleatorizados e, quando existe um caminho, tais algoritmos podem falhar e deixar de informar que tal caminho existe com probabilidade $1/n_c$ para $c > 0$ arbitrária. Os tempos de consulta são $O(1)$ com alta probabilidade. Para fecho transitivo, o tempo amortizado para cada atualização é $O(V_{4/3} \lg^{1/3} V)$. Para caminhos mínimos para todos os pares, os tempos de atualização dependem das consultas. Para consultas que dão apenas os pesos dos caminhos mínimos, o tempo amortizado por atualização é $O(V_3/E \lg^2 V)$. O tempo para informar o caminho mínimo propriamente dito é $\min(O(V_{3/2} \sqrt{\lg v}), O(V_3/E \lg^2 V)$. Demetrescu e Italiano [84] mostraram como tratar operações de atualização e consulta quando as arestas são inseridas e também eliminadas, desde que cada aresta dada tenha uma faixa limitada de valores possíveis extraída do conjunto de números reais.

Aho, Hopcroft e Ullman [5] definiram uma estrutura algébrica conhecida como "semianel", que serve como uma estrutura geral para resolver problemas de caminhos em grafos dirigidos. O algoritmo de Floyd-Warshall, bem como o do fecho transitivo da Seção 25.2 são instanciações de um algoritmo para todos os pares baseado em semianéis. Maggs e Plotkin [240] mostraram como encontrar árvores geradoras mínimas usando um semianel.

26 FLUXO MÁXIMO

Da mesma maneira que podemos modelar um mapa rodoviário como um grafo dirigido para encontrar o caminho mínimo de um ponto a outro, também podemos interpretar um grafo dirigido como uma "rede de fluxo" e usá-lo para responder a perguntas sobre fluxos de materiais. Imagine um material percorrendo um sistema desde uma fonte onde o material é produzido até um sorvedouro, onde ele é consumido. A fonte produz o material a alguma taxa fixa, e o sorvedouro consome o material à mesma taxa. O "fluxo" do material em qualquer ponto no sistema é intuitivamente a taxa pela qual o material se move. Redes de fluxo podem modelar muitos problemas, entre eles líquidos que fluem por tubos, peças que percorrem linhas de montagem, correntes que passam por redes elétricas e informações transmitidas por redes de comunicação.

Podemos imaginar cada aresta dirigida em uma rede de fluxo como um conduto para o material. Cada conduto tem uma capacidade estabelecida, dada como uma taxa máxima pela qual o material pode fluir pelo conduto, como 200 litros de líquido por hora por um cano ou 20 ampères de corrente elétrica por um fio condutor. Vértices são junções de condutos e, exceto quando se trata da fonte e do sorvedouro, o material flui pelos vértices sem se acumular. Em outras palavras, a taxa pela qual o material entra em um vértice deve ser igual à taxa pela qual sai do vértice. Denominamos essa propriedade "conservação do fluxo", e ela é equivalente à lei das correntes de Kirchhoff para a qual o material é a corrente elétrica.

No problema de fluxo máximo, desejamos calcular a maior taxa pela qual podemos despachar material da fonte até o sorvedouro sem infringir quaisquer restrições à capacidade. Esse é um dos problemas mais simples relacionados a redes de fluxo e, como veremos neste capítulo, pode ser resolvido por algoritmos eficientes. Além disso, podemos adaptar as técnicas básicas usadas em algoritmos de fluxo máximo para resolver outros problemas de redes de fluxo.

Este capítulo apresenta dois métodos gerais para resolver o problema do fluxo máximo. A Seção 26.1 formaliza as noções de redes de fluxo e fluxos, definindo formalmente o problema do fluxo máximo. A Seção 26.2 descreve o método clássico de Ford e Fulkerson para determinar fluxos máximos. Uma aplicação desse método, encontrar um emparelhamento máximo em um grafo bipartido não dirigido, é dada na Seção 26.3. A Seção 26.4 apresenta o método *push-relabel* (*empurrar–renomear*), que serve de base para muitos dos algoritmos mais rápidos para problemas de redes de fluxo. A Seção 26.5 abrange o algoritmo *relabel-to-front* (*renomear e posicionar à frente*), uma implementação particular do método push–relabel que é executado no tempo

$O(V_3)$. Embora esse não seja o algoritmo mais rápido conhecido, ilustra algumas das técnicas usadas nos algoritmos assintoticamente mais rápidos e é razoavelmente eficiente na prática.

26.1 REDES DE FLUXO

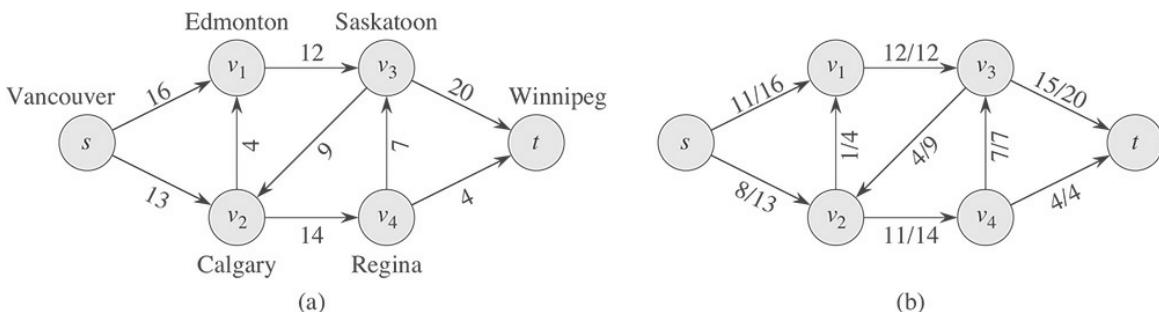
Nesta seção, daremos uma definição para redes de fluxo do ponto de vista da teoria dos grafos, discutiremos suas propriedades e definiremos com exatidão o problema do fluxo máximo. Apresentaremos também, algumas regras úteis de notação.

Redes de fluxo e fluxos

Uma ***rede de fluxo*** $G = (V, E)$ é um grafo dirigido no qual cada aresta $(u, v) \in E$ tem uma ***capacidade*** não negativa $c(u, v) \geq 0$. Impomos ainda mais que, se E contém uma aresta (u,v), então não há nenhuma aresta (v, u) na direção contrária. (Veremos em breve como contornar essa restrição.) Se $(u,v) \notin E$, então, por conveniência, definimos $c(u,v) = 0$, e proibimos laços. Distinguimos dois vértices em uma rede de fluxo: uma ***fonte*** s e um ***sorvedouro*** t. Por conveniência, consideramos que cada vértice se encontra em algum caminho da fonte até o sorvedouro. Isto é, para todo vértice $v \in V$, a rede de fluxo contém um caminho $s \quad v \quad t$. Portanto, o grafo é conexo e, visto que cada vértice exceto s tem no mínimo uma aresta de entrada, $|E| \geq |V| - 1$. A Figura 26.1 mostra um exemplo de rede de fluxo.

Agora, estamos prontos para dar uma definição mais formal de fluxos. Seja $G = (V, E)$ uma rede de fluxo com uma função capacidade c. Seja s a fonte da rede e seja t o sorvedouro. Um ***fluxo*** em G é uma função de valor real f : $V \times V \rightarrow$ que satisfaz as três propriedades seguintes:

Restrição de capacidade: Para todo $u, v \in V$, exigimos $0 \leq f(u, v) \leq c(u, v)$.

Conservação de fluxo: Para todo $u \in V - \{s, t\}$, impomos

$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v) .$$

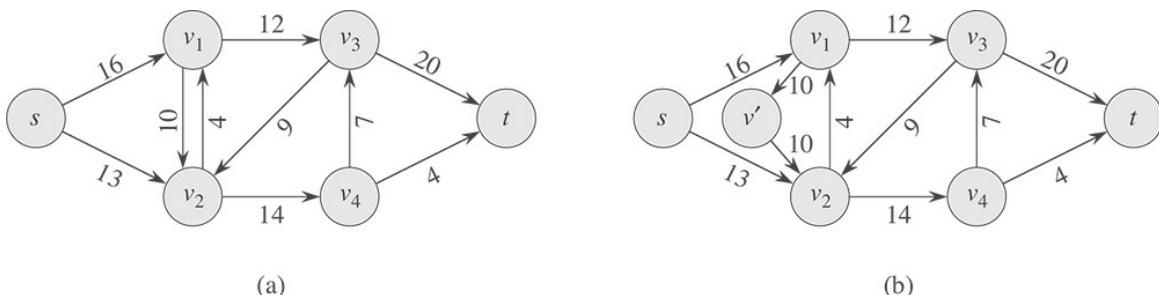
Quando $(u, v) \notin E$, não pode haver nenhum fluxo de u a v, e $f(u,v) = 0$.

A quantidade não negativa $f(u, v)$ é denominada ***fluxo*** do vértice u ao vértice v. O ***valor*** $|f|$ de um fluxo é definido como

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) . \tag{26.1}$$

isto é, o fluxo total que sai da fonte menos o fluxo que entra na fonte. (Aqui, a notação $|\cdot|$ identifica valor de fluxo e não valor absoluto ou cardinalidade.) Normalmente, uma rede de fluxo não terá nenhuma aresta de entrada na fonte, e o fluxo que entra na fonte, dado pelo somatório $\sum_{v \in V} f(v, s)$, será 0. Contudo, nós o incluímos porque, quando apresentarmos redes residuais mais adiante neste capítulo, o fluxo que entra na fonte se tornará significativo. No ***problema do fluxo máximo*** temos uma rede de fluxo G com fonte s e sorvedouro t, e desejamos encontrar um fluxo de valor máximo.

Figura 26.1 (**a**) Uma rede de fluxo $G = (V, E)$ para o problema do transporte da Lucky Puck Company. A fábrica de Vancouver é a fonte s, e o armazém de Winnipeg é o sorvedouro t. A empresa entrega discos para hóquei (*pucks*) em cidades intermediárias, mas somente $c(u, v)$ caixotes por dia podem ir da cidade u para a cidade v. Cada aresta é identificada por sua capacidade. (**b**) Um fluxo f em G com valor $|f| = 19$. Cada aresta (u, v) é identificada por $f(u, v)/c(u, v)$. A barra inclinada na notação serve apenas para separar fluxo e capacidade; não indica divisão.

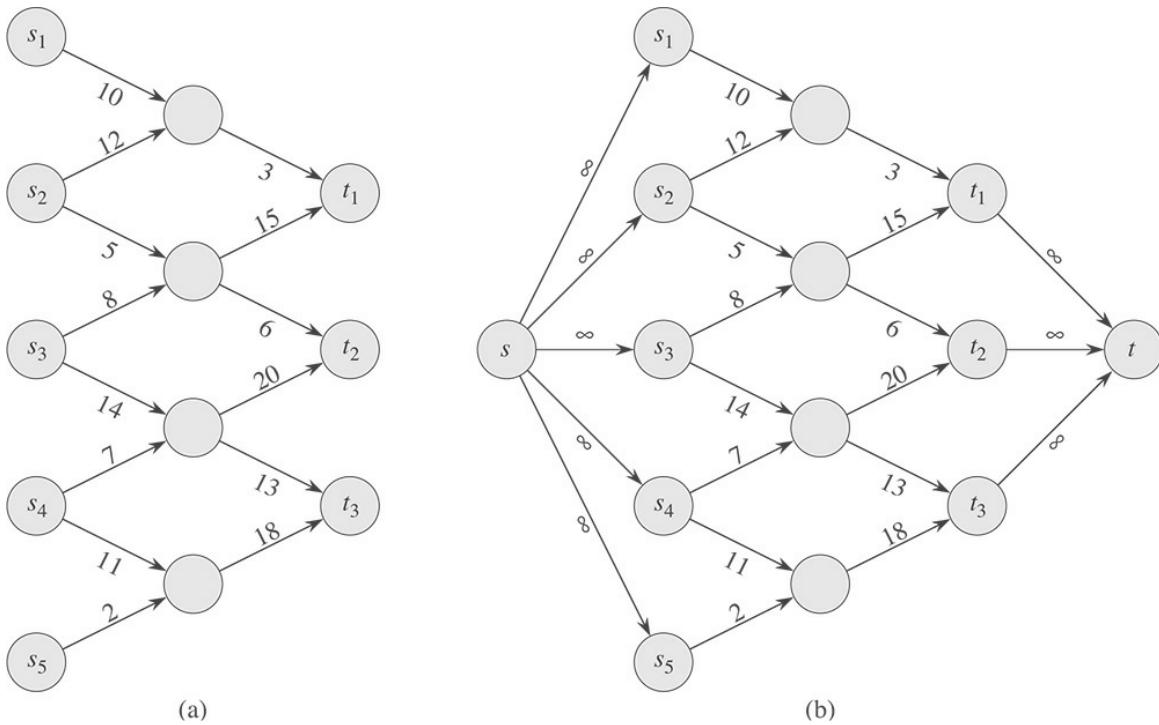
Antes de vermos um exemplo de problema de rede de fluxo, vamos explorar brevemente a definição de fluxo e as duas propriedades de fluxo. A restrição de capacidade diz simplesmente que o fluxo de um vértice a um outro deve ser não negativo e não deve exceder a capacidade dada. A propriedade de conservação de fluxo diz que o fluxo total que entra em um vértice exceto a fonte ou o sorvedouro deve ser igual ao fluxo total que sai do vértice — em linguagem informal, "fluxo que entra é igual a fluxo que sai".

Um exemplo de fluxo

Uma rede de fluxo pode modelar o problema de transporte rodoviário mostrado na Figura 26.1(a). A Lucky Puck Company tem uma fábrica (fonte s) em Vancouver que produz discos para hóquei e um armazém (sorvedouro t) em Winnipeg que os mantém em estoque. A Lucky Puck aluga espaço em caminhões de outra empresa para transportar os discos da fábrica ao armazém. Como os caminhões percorrem rotas especificadas (arestas) entre cidades (vértices) e têm capacidade limitada, a Lucky Puck pode despachar no máximo $c(u, v)$ caixotes por dia entre cada par de cidades u e v na Figura 26.1(a). A Lucky Puck não tem nenhum controle sobre essas rotas e capacidades, e, assim, não pode alterar a rede de fluxo mostrada na Figura 26.1(a). A empresa precisa determinar o maior número p de caixotes que pode despachar por dia e depois produzir essa quantidade, pois não tem sentido produzir mais discos do que é possível transportar para o armazém. A Lucky Puck não está preocupada com o tempo gasto para um determinado disco ir da fábrica ao armazém; o que interessa à empresa é que p caixotes saiam da fábrica por dia e p caixotes cheguem ao armazém por dia.

Podemos modelar o "fluxo" de remessas com um fluxo nessa rede porque o número de caixotes despachados por dia de uma cidade para outra está sujeito a uma restrição de capacidade. Além disso, o modelo deve obedecer à conservação de fluxo porque, em regime estável, a taxa de entrada dos discos em uma cidade intermediária tem de ser igual à taxa de saída dos discos dessa mesma cidade. Caso contrário, os caixotes se acumulariam em cidades intermediárias.

Modelando problemas com arestas antiparalelas

Suponha que a empresa transportadora oferecesse à Lucky Puck a oportunidade de alugar espaço para 10 caixotes em caminhões que fazem a rota Edmonton–Calgary. Seria natural adicionar essa oportunidade ao nosso exemplo e formar a rede mostrada na Figura 26.2(a). Porém, essa rede apresenta um problema: infringe nossa hipótese original, isto é, se uma aresta $(v_1, v_2) \in E$, então $(v_2, v_1) \notin E$. Denominamos as duas arestas (v_1, v_2) ***antiparalelas.*** Assim, se quisermos modelar um problema de fluxo com arestas antiparalelas, temos de transformar a rede em uma outra rede equivalente que não contenha nenhuma aresta antiparalela. A Figura 26.2(b) mostra essa rede equivalente.

Figura 26.2 Conversão de uma rede com arestas antiparalelas em uma rede equivalente sem nenhuma aresta antiparalela. **(a)** Rede de fluxo que contém as arestas (v_1, v_2) e (v_2, v_1). **(b)** Uma rede equivalente sem nenhuma aresta antiparalela. Adicionamos um novo vértice v' e substituímos a aresta (v_1, v_2) pelo par de arestas (v_1, v') e (v', v_2), ambas com a mesma capacidade de (v_1, v_2).

Escolhemos uma das duas arestas antiparalelas, nesse caso (v_1, v_2), e a repartimos adicionando um novo vértice v' e substituindo a aresta (v_1, v_2) pelo par de arestas (v_1, v') e (v', v_2). Também definimos a capacidade das duas novas arestas como a capacidade da aresta original. A rede resultante satisfaz a seguinte propriedade: se uma aresta está na rede, a aresta inversa não está. O Exercício 26.1-1 pede que você prove que a rede resultante é equivalente à original.

Assim, vemos que um problema do mundo real poderia ser modelado muito naturalmente com uma rede de arestas antiparalelas. Todavia, será conveniente desautorizar arestas antiparalelas e, assim, temos um modo direto de converter uma rede que contém arestas antiparalelas em uma rede equivalente sem nenhuma aresta antiparalela.

Redes com várias fontes e vários sorvedouros

Um problema de fluxo máximo pode ter várias fontes e vários sorvedouros, em vez de apenas uma unidade de cada. Por exemplo, na realidade, a Lucky Puck Company poderia ter um conjunto de m fábricas $\{s_1, s_2, ..., s_m\}$ e um conjunto de n armazéns $\{t_1, t_2, ..., t_n\}$, como mostra a Figura 26.3(a). Felizmente, esse problema não é mais difícil que o fluxo máximo comum.

Podemos reduzir o problema de determinar um fluxo máximo em uma rede com várias fontes e vários sorvedouros a um problema de fluxo máximo comum. A Figura 26.3(b) mostra como converter a rede de (a) em uma rede de fluxo comum com somente uma fonte e um sorvedouro. Adicionamos uma ***superfonte*** s e acrescentamos uma aresta dirigida (s, s_i) com capacidade $c(s, s_i) = \infty$ para cada $i = 1, 2, ..., m$. Criamos também um novo ***supersorvedouro*** t e acrescentamos uma aresta dirigida (t_i, t) com capacidade $c(t_i, t) = \infty$ para cada $i = 1, 2, ..., n$. Intuitivamente, qualquer fluxo na rede em (a) corresponde a um fluxo na rede em (b), e *vice-versa*. A única fonte s simplesmente fornece a quantidade de fluxo desejada para as várias fontes s_i, e, igualmente, o único sorvedouro t consome a quantidade de fluxo desejada para os vários sorvedouros t_i. O Exercício 26.1–2 pede que você prove formalmente que os dois problemas são equivalentes.

Figura 26.3 Conversão de um problema de fluxo máximo de várias fontes e vários sorvedouros em um problema com uma única fonte e um único sorvedouro. **(a)** Uma rede de fluxo com cinco fontes $S = \{s_1, s_2, s_3, s_4, s_5\}$ e três sorvedouros $T = \{t_1, t_2, t_3\}$. **(b)** Uma rede de fluxo equivalente com uma única fonte e um único sorvedouro. Adicionamos uma superfonte s e uma aresta com capacidade infinita de s até cada uma das várias fontes. Adicionamos também um supersorvedouro t e uma aresta com capacidade infinita de cada um dos vários sorvedouros a t.

Exercícios

26.1-1 Mostre que repartir uma aresta n em uma rede de fluxo produz uma rede equivalente. Em linguagem mais formal, suponha que a rede de fluxo G contenha uma aresta (u,v) e que criamos uma nova rede de fluxo G' criando um novo vértice x e substituindo (u,v) por novas arestas (u,x) e (x,v) com $c(u,x) = c(x,v) = c(u,v)$. Mostre que um fluxo máximo em G' tem o mesmo valor que um fluxo máximo em G.

26.1-2 Estenda as propriedades e definições de fluxo ao problema de várias fontes e vários sorvedouros. Mostre que qualquer fluxo em uma rede de fluxo com várias fontes e vários sorvedouros corresponde a um fluxo de valor idêntico na rede de fonte única e sorvedouro único obtida pela adição de uma superfonte e um supersorvedouro, e vice-versa.

26.1-3 Suponha que uma rede de fluxo $G = (V,E)$ transgrida a hipótese de que a rede contém um caminho $s \quad v \quad t$ para todos os vértices $v \in V$. Seja u um vértice para o qual não há nenhum caminho $s \quad u \quad t$. Mostre que deve existir um fluxo máximo f em G tal que $f(u, v) = f(v, u) = 0$ para todos os vértices $v \in V$.

26.1-4 Seja f um fluxo em uma rede e seja a um número real. O ***múltiplo escalar do fluxo*** denotado por af é uma função de $V \times V$ para definida por

$(af)(u, v) = a \cdot f(u, v)$.

Prove que os fluxos em uma rede formam um ***conjunto convexo***. Isto é, mostre que, se f_1 e f_2 são fluxos, então $af_1 + (1 - a)f_2$ também é um fluxo para todo a no intervalo $0 \leq a \leq 1$.

26.1-5 Enuncie o problema de fluxo máximo como um problema de programação linear.

26.1-6 O professor Adam tem dois filhos que, infelizmente, não gostam um do outro. O problema é tão grave que eles não só se recusam a ir à escola juntos mas, na verdade, cada um se recusa a passar por qualquer quadra pela qual o outro tenha passado naquele dia. Porém, eles não se importam se seus caminhos se cruzarem em uma esquina. Felizmente, a casa do professor, bem como a escola, está situada em esquina mas, fora isso, ele não tem certeza de que será possível enviar os filhos à mesma escola. O professor tem um mapa da cidade. Mostre como formular o problema de determinar se os dois filhos do professor podem frequentar a mesma escola como um problema de fluxo máximo.

26.1-7 Suponha que, além das capacidades de arestas, uma rede de fluxo tenha ***capacidades de vértices.*** Isto é, cada vértice v tem um limite $l(v)$ para a quantidade de fluxo que pode passar por v. Mostre como transformar uma rede de fluxo $G = (V,E)$ com capacidades de vértices em uma rede de fluxo equivalente $G' = (V',E')$ sem capacidade de vértices, tal que um fluxo máximo em G' tenha o mesmo valor que um fluxo máximo em G. Quantos vértices e quantas arestas G' tem?

26.2 O método Ford-Fulkerson

Esta seção apresenta o método de Ford e Fulkerson para resolver o problema do fluxo máximo. Nós o denominamos "método" em vez de "algoritmo" porque ele abrange diversas implementações com diferentes tempos de execução. O método Ford-Fulkerson depende de três ideias importantes que transcendem o método e são relevantes para muitos algoritmos e problemas de fluxo: redes residuais, caminhos aumentadores e cortes. Essas ideias são essenciais para o importante teorema do fluxo máximo/corte mínimo (Teorema 26.6), que caracteriza o valor de um fluxo máximo em termos de cortes da rede de fluxo. Encerramos esta seção apresentando uma implementação específica do método Ford-Fulkerson e analisando seu tempo de execução.

O método Ford-Fulkerson aumenta iterativamente o valor do fluxo. Começamos com $f(u, v) = 0$ para todo $u, v \in V$, que dá um fluxo inicial de valor 0. A cada iteração, aumentamos o valor do fluxo em G determinando um "caminho aumentador" em uma "rede residual" G_f. associada. Tão logo conheçamos as arestas de um caminho aumentador em G_f, fica fácil identificar arestas específicas em G cujo fluxo podemos mudar e, com isso, aumentar o valor do fluxo. Embora cada iteração do método de Ford-Fulkerson aumente o valor do fluxo, veremos que o fluxo em qualquer aresta particular de G pode aumentar ou diminuir; pode ser necessário diminuir o fluxo em algumas arestas para permitir que um algoritmo envie mais fluxo da fonte ao sorvedouro. Aumentamos repetidamente o fluxo até que a rede residual não tenha mais caminhos aumentadores. O teorema do fluxo máximo/corte mínimo mostrará que, no término, esse processo produz um fluxo máximo.

```
FORD-FULKERSON-METHOD(G, s, t)
1  inicializar fluxo f como 0
2  while existir um caminho aumentador p na rede residual G_f
3     aumentar fluxo f ao longo de p
4  return f
```

Redes residuais

Intuitivamente, dados uma rede de fluxo G e um fluxo f, a rede residual G_f consiste em arestas com capacidades que representam como podemos mudar o fluxo em arestas de G. Uma aresta da rede de fluxo pode admitir uma quantidade de fluxo adicional igual à capacidade da aresta menos o fluxo nessa aresta. Se tal valor é positivo, colocamos essa aresta em G_f com uma "capacidade residual" de $c_f(u, v) = c(u, v) - f(u, v)$. As únicas arestas de G que estão em G_f são as que podem receber mais fluxo; as arestas (u,v) cujos fluxos são iguais às suas capacidades têm $c_f(u,v) = 0$ e não estão em G_f.

Porém, a rede residual G_f também pode conter arestas que não estão em G. Quando um algoritmo é aplicado a um fluxo com a finalidade de aumentar o fluxo total, pode ser necessário reduzir o fluxo em determinada aresta. Para representar uma possível diminuição de um fluxo positivo $f(u, v)$ em uma aresta em G, inserimos uma aresta (v, u) em G_f com capacidade residual $c_f(v, u) = f(u, v)$ — isto é, uma aresta que pode admitir fluxo na direção oposta a (u, v) e, no máximo, eliminar o fluxo em (u, v).

Essas arestas invertidas na rede residual permitem que um algoritmo envie de volta o fluxo que já enviou ao longo de uma aresta. Enviar o fluxo de volta ao longo de uma aresta equivale a *diminuir o* fluxo na aresta, que é uma operação necessária em muitos algoritmos.

Em linguagem mais formal, suponha que tenhamos uma rede de fluxo $G = (V,E)$ com fonte s e sorvedouro t. Seja f um fluxo em G, e considere um par de vértices $u, v \in V$. Definimos a ***capacidade residual*** $c_f(u, v)$ por

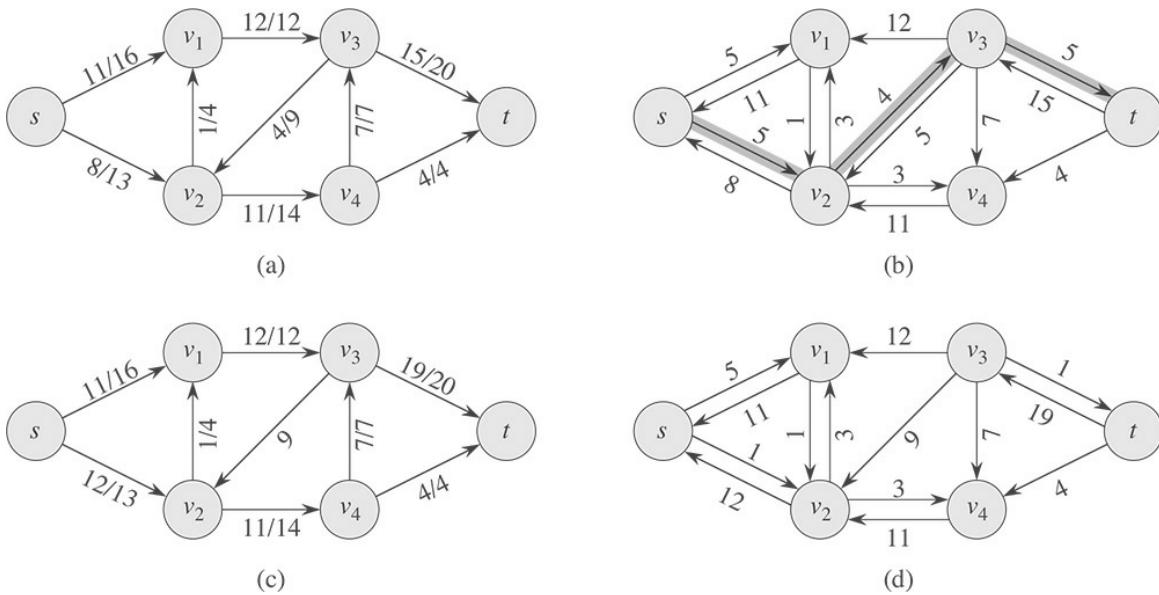
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{se } (u,v) \in E \text{ ,} \\ f(v,u) & \text{se } (v,u) \in E \text{ ,} \\ 0 & \text{caso contrário .} \end{cases} \tag{26.2}$$

Como supomos que $(u, v) \in E$ implica $(v, u) \notin E$, exatamente um caso na equação (26.2) se aplica a cada par ordenado de vértices.

Como exemplo da equação (26.2), se $c(u, v) = 16$ e $f(u, v) = 11$, então podemos aumentar $f(u, v)$ de até $c_f(u, v)$ = cinco unidades antes de infringir a restrição de capacidade da aresta (u, v). Queremos também permitir que um algoritmo retorne até 11 unidades de fluxo de v a u e, por consequência, $c_f(v, u) = 11$.

Dada uma rede de fluxo $G = (V, E)$ e um fluxo f, a ***rede residual*** de G induzida por f é $G_f = (V, E_f)$, onde

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\} \text{ .} \tag{26.3}$$

Isto é, como prometido, cada aresta da rede residual, ou ***aresta residual***, pode admitir um fluxo que é maior do que 0. A Figura 26.4(a) repete a rede de fluxo G e fluxo f da Figura 26.1(b), e a Figura 26.4(b) mostra a rede residual G_f correspondente. As arestas em E_f são arestas em E ou são suas inversas e, assim,

$$|E_f| \leq 2\,|E|.$$

Observe que a rede residual G_f é semelhante a uma rede de fluxo com capacidades dadas por c_f. Ela não satisfaz nossa definição de rede de fluxo porque pode conter uma aresta (u, v), bem como sua inversa, (v, u). Fora essa diferença, uma rede residual tem as mesmas propriedades que uma rede de fluxo, e podemos definir um fluxo na rede residual como um fluxo que satisfaz a definição de um fluxo, porém em relação às capacidades c_f na rede G_f.

Um fluxo em uma rede residual nos dá um guia para adicionar fluxo à rede de fluxo original. Se f é um fluxo em G e f' é um fluxo na rede residual G_f correspondente, definimos $f \uparrow f'$, o ***aumento*** do fluxo f de f', como a função de $V \times V$ em , definida por

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{se } (u,v) \in E, \\ 0 & \text{caso contrário}. \end{cases} \tag{26.4}$$

A intuição que fundamenta essa definição decorre da definição da rede residual. Aumentamos o fluxo em (u, v) de $f'(u, v)$ mas o reduzimos de $f'(u, v)$ porque empurrar fluxo pela aresta invertida na rede residual significa reduzir o fluxo na rede original. Empurrar fluxo pela aresta invertida na rede residual é uma operação também conhecida como ***cancelamento***. Por exemplo, enviar cinco caixotes de discos de hóquey de u a v e enviar dois caixotes de v a u seria o mesmo (da perspectiva do resultado final) que enviar três caixotes de u a v e nenhum de v a u. Cancelamento desse tipo é crucial para qualquer algoritmo de fluxo máximo.

Figura 26.4 **(a)** Rede de fluxos G e f da Figura 26.1(b). **(b)** Rede residual Gf com caminho aumentador p sombreado; sua capacidade residual é $c_f(p) = c_f(v_2, v_3) = 4$. Arestas com capacidade residual igual a 0, como (v_2, v_3), não são mostradas, uma convenção que seguiremos no restante desta seção. **(c)** O fluxo em G que resulta de aumentar essa rede ao logo do caminho p de uma quantidade equivalente à sua capacidade residual, 4. Arestas que não transportam nenhum fluxo, como (v_3, v_2), são identificadas somente por suas capacidades, uma outra convenção que seguimos neste capítulo. **(d)** Rede residual induzida pelo fluxo em (c).

Lema 26.1

Seja $G = (V, E)$ uma rede de fluxo com fonte s e sorvedouro t, e seja f um fluxo em G. Seja G_f a rede residual de G induzida por f e seja f' um fluxo em G_f. Então, a função $f \uparrow f'$ definida na equação (26.4) é um fluxo em G com valor $|f \uparrow f'| = |f| + |f'|$.

Prova Em primeiro lugar, temos de verificar se $f \uparrow f'$ obedece às restrições de capacidade para cada aresta em E e conservação de fluxo em cada vértice em $V - \{s, t\}$.

Para a restrição de capacidade, observe em primeiro lugar que, se $(u, v) \in E$, então $c_f(v, u) = f(u, v)$. Portanto, temos $f'(v, u) \leq c_f(v, u) = f(u, v)$, e, por consequência,

$$
\begin{aligned}
(f \uparrow f)(u,v) &= f(u,v) + f'(u,v) - f'(u,v) && \text{(pela equação (26.4))} \\
&\geq f(u,v) + f'(u,v) - f(u,v) && \text{(porque } f'(u,v) \leq f(u,v)\text{)} \\
&= f'(u,v) \\
&\geq 0 \, ,
\end{aligned}
$$

Além disso, $(f \uparrow f')\,(u, v)$

$$
\begin{aligned}
(f \uparrow f)\,(u,v) & \\
&= f(u,v) + f'(u,v) - f'(u,v) && \text{(pela equação (26.4))} \\
&\leq f(u,v) + f'(u,v) && \text{(porque os fluxos são não negativos)} \\
&\leq f(v,u) + c_f(u,v) && \text{(restrição de capacidade)} \\
&= f(u,v) + c(u,v) - f(u,v) && \text{(definição de } c_f\text{)} \\
&= c(u,v)
\end{aligned}
$$

Para conservação de fluxo, como f e f' obedecem à conservação de fluxo, temos que, para todo $u \in V - \{s, t\}$,

$$
\begin{aligned}
\sum_{v \in V} (f \uparrow f')(u,v) &= \sum_{v \in V} (f(u,v) + f'(u,v) - f'(v,u)) \\
&= \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) - \sum_{v \in V} f'(v,u) \\
&= \sum_{v \in V} f(v,u) + \sum_{v \in V} f'(v,u) - \sum_{v \in V} f'(u,v) \\
&= \sum_{v \in V} (f(v,u) + f'(v,u) - f'(u,v)) \\
&= \sum_{v \in V} (f \uparrow f')(v,u) \, ,
\end{aligned}
$$

onde a terceira linha decorre da segunda por conservação de fluxo.

Finalmente, calculamos o valor de $f \uparrow f'$. Lembre-se de que desautorizamos arestas antiparalelas em G (mas não em G_f) e, por consequência, para cada vértice $v \in V$, sabemos que pode haver uma aresta (s, v) ou (v, s), mas nunca ambas. Definimos $V_1 = \{v : (s, v) \in E\}$ como o conjunto de vértices com arestas que saem de s e $V_2 = \{v : (v, s) \in E\}$ como o conjunto de vértices com arestas que se dirigem a s. Temos $V_1 \cup V_2 \subseteq V$ e, como desautorizamos arestas antiparalelas, $V_1 \cap V_2 = \emptyset$. Agora calculamos

$$
\begin{aligned}
|f \uparrow f'| &= \sum_{v \in V} (f \uparrow f')(s,v) - \sum_{v \in V} (f \uparrow f')(v,s) \\
&= \sum_{v \in V_1} (f \uparrow f')(s,v) - \sum_{v \in V_2} (f \uparrow f')(v,s) \, ,
\end{aligned}
\tag{26.5}
$$

onde a segunda linha decorre de $(f \uparrow f')(w, x)$ ser 0 se $(w, x) \notin E$. Agora aplicamos a definição de $f \uparrow f'$ à equação (26.5) e reordenamos e agrupamos termos para obter

$$
\begin{aligned}
|f \uparrow f'| \\
&= \sum_{v \in V_1} (f(s,v) + f'(s,v) - f'(v,s)) - \sum_{v \in V_2} (f(v,s) + f'(v,s) - f'(s,v)) \\
&= \sum_{v \in V_1} f(s,v) + \sum_{v \in V_1} f'(s,v) - \sum_{v \in V_1} f'(v,s) \\
&\qquad - \sum_{v \in V_2} f(v,s) - \sum_{v \in V_2} f'(v,s) + \sum_{v \in V_2} f'(s,v) \\
&= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) \\
&\qquad + \sum_{v \in V_1} f'(s,v) + \sum_{v \in V_2} f'(s,v) - \sum_{v \in V_1} f'(v,s) - \sum_{v \in V_2} f'(v,s) \\
&= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) + \sum_{v \in V_1 \cup V_2} f'(s,v) - \sum_{v \in V_1 \cup V_2} f'(v,s) \,. \qquad (26.6)
\end{aligned}
$$

Na equação (26.6), podemos estender todos os quatro somatórios para V, já que cada termo adicional tem valor 0. (O Exercício 26.2-1 pede que você prove isso formalmente.) Assim, temos

$$
\begin{aligned}
|f \uparrow f'| &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{v \in V} f'(s,v) - \sum_{v \in V} f'(v,s) \\
&= |f| + |f'| \,. \qquad (26.7)
\end{aligned}
$$

Caminhos aumentadores

Dados uma rede de fluxo $G = (V, E)$ e um fluxo f, um ***caminho aumentador*** p é um caminho simples de s a t na rede residual G_f. Pela definição da rede residual, podemos aumentar o fluxo em uma aresta (u, v) de um caminho aumentador até $c_f(u, v)$ sem infringir a restrição de capacidade imposta a qualquer (u, v) e (v, u) que está na rede de fluxo original G.

O caminho sombreado na Figura 26.4(b) é um caminho aumentador. Tratando a rede residual G_f na figura como uma rede de fluxo, podemos aumentar o fluxo que percorre cada aresta desse caminho de até quatro unidades sem infringir a restrição de capacidade, já que a menor capacidade residual nesse caminho é $c_f(v_2, v_3) = 4$. A quantidade máxima possível de aumento para o fluxo em cada aresta de um caminho aumentador p é denominada ***capacidade residual*** de p e é dada por

$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ está em } p\}$.

O lema a seguir, cuja prova deixamos para o Exercício 26.2-7, torna esse argumento mais preciso.

Lema 26.2

Seja $G = (V, E)$ uma rede de fluxo, seja f um fluxo em G e seja p um caminho aumentador em G_f. Defina uma função $f_p : V \times V \to$ por

$$
f_p(u,v) = \begin{cases} c_f(p) & \text{se } (u,v) \text{ está em } p, \\ 0 & \text{caso contrário} \,. \end{cases} \qquad (26.8)
$$

Então, f_p é um fluxo em G_f com valor $|f_p| = c_f(p) > 0$.

O corolário a seguir mostra que, se aumentarmos f adicionando f_p, obtemos um outro fluxo em G cujo valor está mais próximo do máximo. A Figura 26.4(c) mostra o resultado de aumentar o fluxo f da Figura 26.4(a) adicionando o fluxo f_p da Figura 26.4(b), e a Figura 26.4(d) mostra a rede residual resultante.

Corolário 26.3

Seja $G = (V, E)$ uma rede de fluxo, seja f um fluxo em G e seja p um caminho aumentador em G_f. Seja f_p definido como na equação (26.8) e suponha que aumentamos f adicionando f_p. Então, a função $f \uparrow f_p$ é um fluxo em G com valor $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Prova Imediata, pelos Lemas 26.1 e 26.2.

Cortes de redes de fluxo

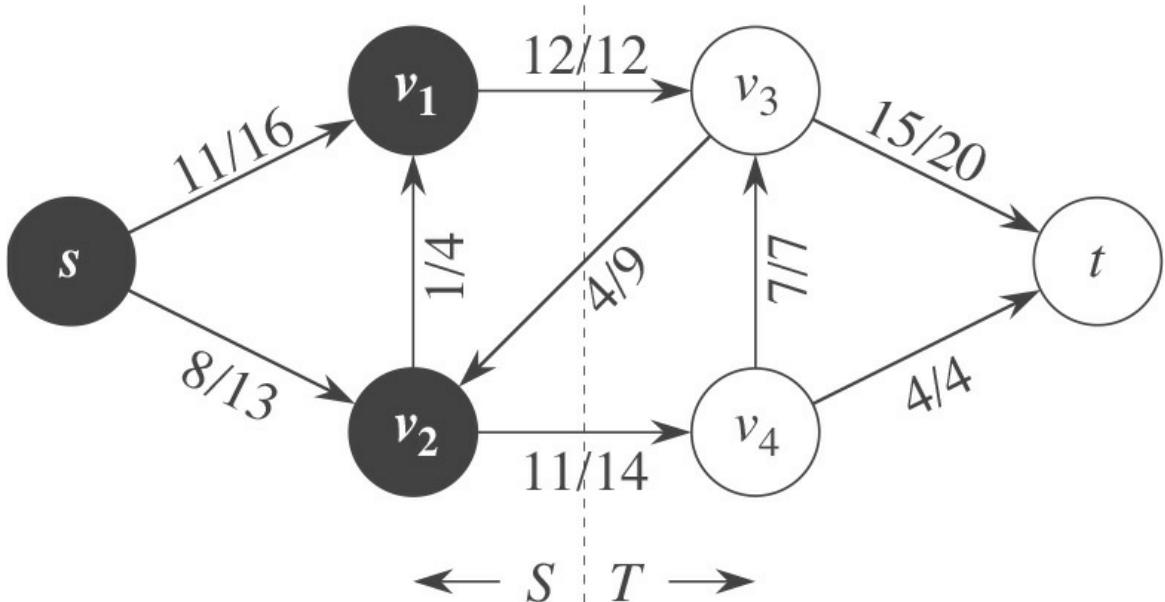
O método Ford-Fulkerson aumenta repetidamente o fluxo ao longo de caminhos aumentadores até encontrar um fluxo máximo. Como sabemos que, quando o algoritmo termina, realmente encontramos um fluxo máximo? O teorema do fluxo máximo/corte mínimo, que demonstraremos em breve, nos diz que um fluxo é máximo se e somente se sua rede residual não contém nenhum caminho aumentador. Entretanto, para provar esse teorema, devemos primeiro explorar a noção de corte de uma rede de fluxo.

Um ***corte*** (S, T) de uma rede de fluxo $G = (V, E)$ é uma partição de V em S e $T = V - S$ tal que $s \in S$ e $t \in T$. (Essa definição é semelhante à definição de "corte" que usamos para árvores geradoras mínimas no Capítulo 23, exceto que aqui estamos cortando um grafo dirigido em vez de um grafo não dirigido, e insistimos que $s \in S$ e $t \in T$.) Se f é um fluxo, então o ***fluxo líquido*** $f(S, T)$. que passa pelo corte é definido como

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(u,v). \qquad (26.9)$$

A ***capacidade*** do corte (S, T) é

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) \; . \qquad (26.10)$$

Um ***corte mínimo*** de uma rede é um corte cuja capacidade é mínima em relação a todos os cortes da rede.

A assimetria entre as definições de fluxo e a capacidade de um corte é intencional e importante. Quando se trata de capacidade, contamos somente as capacidades das arestas que vão de S a T, ignorando arestas na direção inversa. Quando se trata de fluxo, consideramos o fluxo de S a T menos o fluxo na direção inversa de T a S.

A razão para essa diferença ficará clara mais adiante nesta seção.

A Figura 26.5 mostra o corte $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ na rede de fluxo da Figura 26.1(b). O fluxo líquido por esse corte é

$$\begin{aligned} f(v_1, v_3) + f(v_2,v_4) - f(v_3,v_2) &= 12 + 11 - 4 \\ &= 19 \, , \end{aligned}$$

Figura 26.5 Um corte (S, T) na rede de fluxo da Figura 26.1(b), onde $S = \{s, v_1, v_2\}$ e $T = \{v_3, v_4, t\}$. Os vértices em S são pretos, e os vértices em T são brancos. O fluxo líquido por (S, T) é $f(S, T) = 19$, e a capacidade é $c(S, T) = 26$.

O lema a seguir mostra que, para um fluxo f dado, o fluxo líquido por qualquer corte é o mesmo, e é igual a $|f|$, o valor do fluxo.

Lema 26.4

Seja f um fluxo de uma rede de fluxo G com fonte s e sorvedouro t, e seja (S, T) qualquer corte de G. Então, o fluxo líquido por (S, T) é $f(S, T) = |f|$.

Prova Podemos reescrever a condição de conservação de fluxo para qualquer nó $u \in V - \{s, t\}$ como

$$\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) = 0. \tag{26.11}$$

Adotando a definição de $|f|$ dada pela equação (26.1) e somando o lado esquerdo da equação (26.11), que é igual a 0, o somatório para todos os vértices em $S - \{s\}$ dá

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) \right).$$

Expandindo o somatório do lado direito e reagrupando termos, obtemos

$$\begin{aligned} |f| &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S - \{s\}} \sum_{v \in V} f(u,v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v,u) \\ &= \sum_{v \in V} \left(f(s,v) + \sum_{u \in S - \{s\}} f(u,v) \right) - \sum_{v \in V} \left(f(v,s) + \sum_{u \in S - \{s\}} f(v,u) \right) \\ &= \sum_{v \in V} \sum_{u \in S} f(u,v) - \sum_{v \in V} \sum_{u \in S} f(v,u) . \end{aligned}$$

Como $V = S \cup T$ e $S \cap T = 0/$, podemos separar cada somatório em V em somatórios em S e T para obter

$$
\begin{aligned}
|f| &= \sum_{v\in S}\sum_{u\in S} f(u,v) + \sum_{v\in T}\sum_{u\in S} f(u,v) - \sum_{v\in S}\sum_{u\in S} f(v,u) - \sum_{v\in T}\sum_{u\in S} f(v,u) \\
&= \sum_{v\in T}\sum_{u\in S} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u) \\
&\qquad + \left(\sum_{v\in S}\sum_{u\in S} f(u,v) - \sum_{v\in S}\sum_{u\in S} f(v,u)\right).
\end{aligned}
$$

Os dois somatórios entre parênteses são, na verdade, iguais, já que para todos os vértices $x, y \in S$ o termo $f(x, y)$ aparece uma vez em cada somatório. Por consequência, esses somatórios cancelam um ao outro e temos

$$
\begin{aligned}
|f| &= \sum_{u\in S}\sum_{v\in T} f(v,u) - \sum_{u\in S}\sum_{v\in T} f(v,u) \\
&= f(S,T)\ .
\end{aligned}
$$

Um corolário para o Lema 26.4 mostra como podemos usar capacidades de corte para limitar o valor de um fluxo.

Corolário 26.5

O valor de qualquer fluxo f em uma rede de fluxo G é limitado por cima pela capacidade de qualquer corte de G.

Prova Seja (S, T) qualquer corte de G e seja f qualquer fluxo. Pelo Lema 26.4 e pela restrição de capacidade,

$$
\begin{aligned}
|f| &= f(S,T) \\
&= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \\
&\leq \sum_{u\in S}\sum_{v\in T} f(u,v) \\
&\leq \sum_{u\in S}\sum_{v\in T} c(u,v) \\
&= c(S,T)\ .
\end{aligned}
$$

O Corolário 26.5 produz a seguinte consequência imediata: o valor de um fluxo máximo em uma rede é limitado por cima pela capacidade de um corte mínimo da rede. O importante teorema de fluxo máximo/corte mínimo que enunciamos e provamos agora diz que o valor de um fluxo máximo é de fato igual à capacidade de um corte mínimo.

Teorema 26.6 (Teorema do fluxo máximo/corte mínimo)

Se f é um fluxo em uma rede de fluxo $G = (V, E)$ com fonte s e sorvedouro t, então as seguintes condições são equivalentes:

1. f é um fluxo máximo em G.
2. A rede residual G_f não contém nenhum caminho aumentador.
3. $|f| = c(S, T)$ para algum corte (S, T) de G.

Prova (1) ⇒ (2): Suponha, por contradição, que f seja um fluxo máximo em G, mas que G_f tenha um caminho aumentador p. Então, pelo Corolário 26.3, o fluxo encontrado aumentando f com a adição de f_p, onde f_p é dado pela equação (26.8), é um fluxo em G com valor estritamente maior que $|f|$, o que contradiz a hipótese de que f seja um fluxo máximo.

(2)⇒ (3): Suponha que G_f não tenha nenhum caminho aumentador, isto é, que G_f não contenha nenhum caminho de s a t. Defina

$$S = \{v \in V : \text{existe um caminho de } s \text{ a } v \text{ em } G_f\}$$

e $T = V - S$. A partição (S, T) é um corte: temos $s \in S$ trivialmente e $t \notin S$ porque não existe nenhum caminho de s a t em G_f. Agora considere um par de vértices $u \in S$ e $v \in T$. Se $(u,v) \in E$, devemos ter $f(v,u) = c(v,u)$, já que, caso contrário, $(u,v) \in E_f$, o que colocaria v no conjunto S. Se $(v, u) \in E$, devemos ter $f(v,u) = 0$, porque, caso contrário, $c_f(u,v) = f(v,u)$ seria positivo e teríamos $(u, v) \in E_f$, o que colocaria v em S. É claro que, se nem (u,v) nem (v,u) está em E, então $f(u,v) = f(v, u) = 0$. Assim, temos

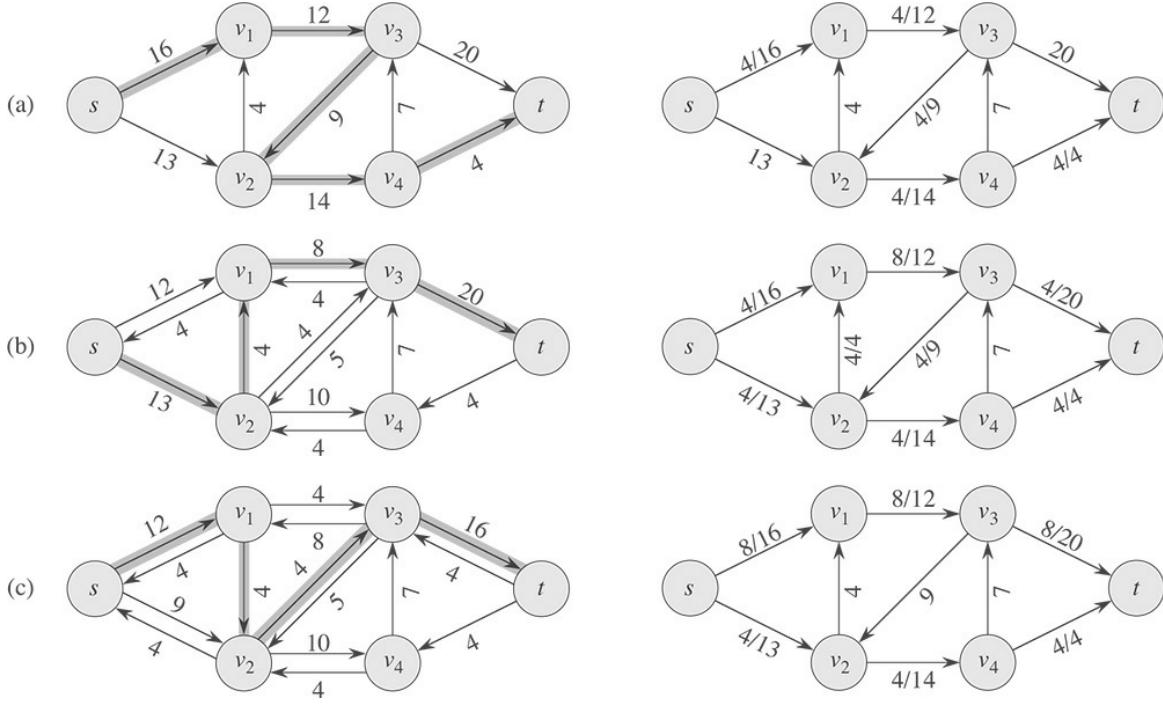
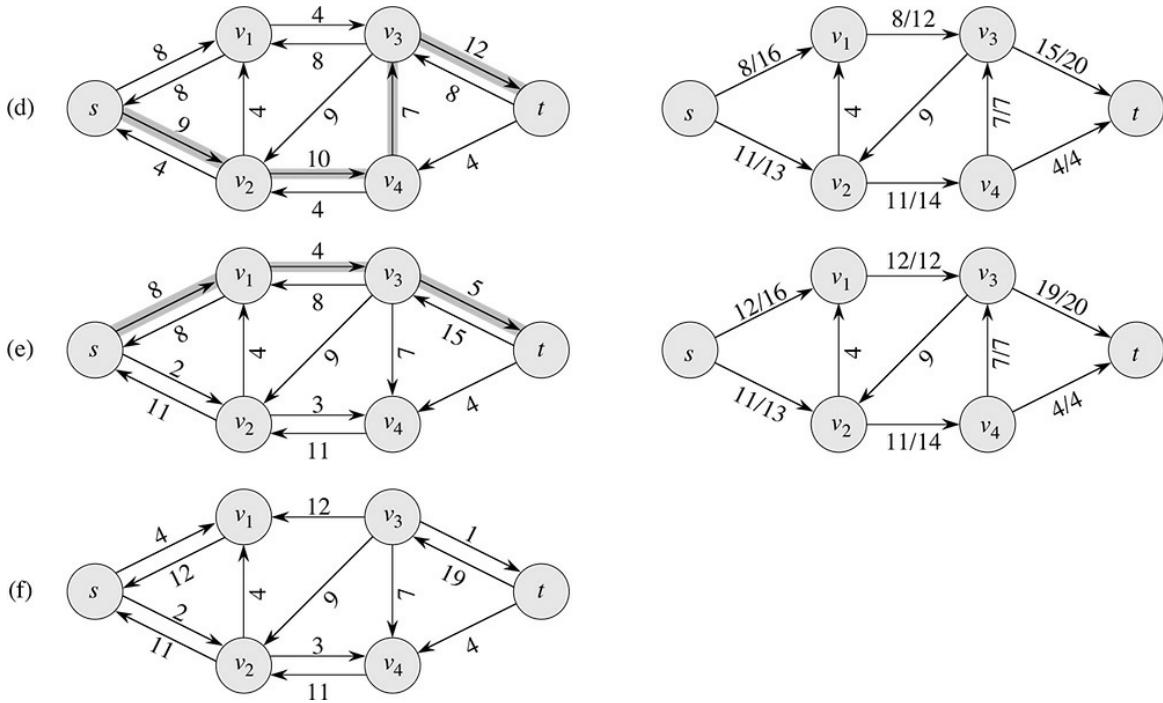
$$\begin{aligned} f(S,T) &= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u) \\ &= \sum_{u\in S}\sum_{v\in T} c(u,v) - \sum_{v\in T}\sum_{u\in S} 0 \\ &= c(S,T)\,. \end{aligned}$$

Portanto, pelo Lema 26.4, $|f| = f(S, T) = c(S, T)$.

(3) ⇒ (1): Pelo Corolário 26.5, $|f|$ $c(S, T)$ para todos os cortes (S, T). Assim, a condição $|f| = c(S, T)$ implica que f seja um fluxo máximo.

O algoritmo básico de Ford e Fulkerson

Em cada iteração do método Ford-Fulkerson, encontramos *algum* caminho aumentador p e usamos p para modificar o fluxo f. Como sugerem o Lema 26.2 e o Corolário 26.3, substituímos f por $f \uparrow f_p$ e obtemos um novo fluxo cujo valor é $|f| + |f_p|$. A implementação do método apresentada a seguir calcula o fluxo máximo em uma rede de fluxo $G = (V,E)$ atualizando o atributo de fonte $(u,v).f$ para cada aresta $(u,v) \in E$.[1]

Se $(u,v) \notin E$, supomos implicitamente que $(u,v).f = 0$. Consideramos também que temos as capacidades $c(u,v)$ juntamente com a rede de fluxo, e $c(u,v) = 0$ se $(u,v) \neq E$. Calculamos a capacidade residual $c_f(u,v)$ de acordo com a fórmula (26.2). A expressão $c_f(p)$ no código serve apenas como uma variável temporária que armazena a capacidade residual do caminho p.

```
FORD-FULKERSON(G, s, t)
1  for cada aresta (u, v) ∈ G.E
2     (u, v).f = 0
3  while existir um caminho p de s a t na rede residual G_f
4     c_f(p) = min{c_f(u, v) : (u, v) está em p}
5     for cada aresta (u, v) em p
6          if (u,v) ∈ E
7                (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```

O algoritmo de FORD-FULKERSON simplesmente expande o pseudocódigo FORD-FULKER-SON-METHOD dado antes. A Figura 26.6 mostra o resultado de cada iteração em um exemplo de execução. As linhas 1–2 inicializam o fluxo f como 0. O laço **while** das linhas 3–8 encontra repetidamente um caminho aumentador p em G_f e aumenta o fluxo f ao longo de p adicionando a capacidade residual $c_f(p)$. Cada aresta residual no caminho p é uma aresta na rede original ou é a inversa de uma aresta na rede original . As linhas 6–8 atualizam o fluxo adequadamente em cada caso, adicionando fluxo quando a aresta residual é uma aresta original e subtraindo, caso contrário. Quando não existe nenhum caminho aumentador, o fluxo f é um fluxo máximo.

Figura 26.6 Execução do algoritmo básico de Ford-Fulkerson. (**a**)-(**e**) Iterações sucessivas do laço **while**. O lado esquerdo de cada parte mostra a rede residual G_f da linha 3 com um caminho aumentador sombreado *p*. O lado direito de cada parte mostra o novo fluxo *f* que resulta do aumento de *f* pela adição de f_p. A rede residual em (a) é a rede de entrada *G*.

Figura 26 .6, **(continuação)** (**f**) Rede residual no último teste do laço **while**. Ela não tem nenhum caminho aumentador e, então, o fluxo *f* mostrado em (e) é um fluxo máximo. O valor do fluxo máximo encontrado é 23.

Análise de Ford-Fulkerson

O tempo de execução de FORD-FULKERSON depende de como determinamos o caminho aumentador p na linha 3. Se o escolhermos mal, o algoritmo pode nem mesmo terminar: o valor do fluxo aumentará com os aumentos sucessivos, mas não precisa sequer convergir para o valor de fluxo máximo.[2] Porém, se determinarmos o caminho aumentador usando uma busca em largura (que vimos na Seção 22.2), o algoritmo será executado em tempo polinomial. Antes de provar esse resultado, obtemos um limite simples para o caso no qual escolhemos o caminho aumentador arbitrariamente e todas as capacidades são inteiras.

Na prática, as capacidades que aparecem nos problemas de fluxo máximo costumam ser números inteiros. Se as capacidades são números racionais, podemos aplicar uma transformação de escala adequada para transformá-los em números inteiros. Se f^* denotar um fluxo máximo na rede transformada, então uma implementação direta de FORD-FULKERSON executa o laço **while** das linhas 3–8 no máximo $|f^*|$ vezes, já que o valor do fluxo aumenta de, no mínimo, uma unidade em cada iteração.

Podemos realizar eficientemente o trabalho executado dentro do laço **while** se implementarmos a rede de fluxo $G = (V, E)$ com a estrutura de dados correta e encontrarmos um caminho aumentador por um algoritmo de tempo linear. Vamos supor que mantemos uma estrutura de dados correspondente a um grafo dirigido $G' = (V, E')$, onde $E' = \{(u, v) : (u, v) \in E \text{ ou } (v, u) \in E\}$. As arestas na rede G também são arestas em G' e, portanto, é fácil manter capacidades e fluxos nessa estrutura de dados. Dado um fluxo f em G, as arestas na rede residual G_f consistem em todas as arestas (u, v) de G' tais que $c_f(u,v) > 0$, onde c_f está de acordo com a equação (26.2). Portanto, o tempo para encontrar um caminho em uma rede residual é $O(V + E') = O(E)$ se usarmos busca em profundidade ou busca em largura. Assim, cada iteração do laço **while** demora o tempo $O(E)$, bem como a inicialização nas linhas 1–2, o que resulta no tempo total de execução $O(E\,|f^*|)$ para o algoritmo de FORD-FULKERSON .

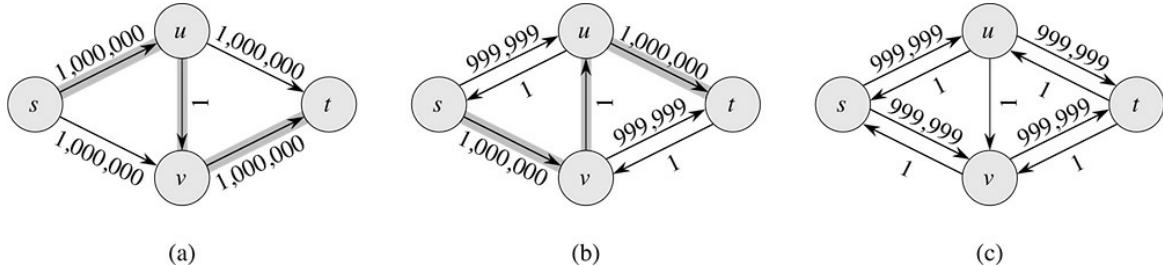
Quando as capacidades são números inteiros e o valor de fluxo ótimo $|f^*|$ é pequeno, o tempo de execução do algoritmo de Ford-Fulkerson é bom. A Figura 26.7(a) mostra um exemplo do que pode acontecer em uma rede de fluxo simples para a qual $|f^*|$ é grande. Um fluxo máximo nessa rede tem valor 2.000.000: 1.000.000 unidades de fluxo que percorrem o caminho $s \to u \to t$, outras 1.000.000 unidades percorrem o caminho $s \to v \to t$. Se o primeiro caminho aumentador encontrado por FORD-FULKERSON é $s \to u \to v \to t$, mostrado na Figura 26.7(a), o fluxo tem valor 1 após a primeira iteração. A rede residual resultante é mostrada na Figura 26.7(b). Se a segunda iteração encontra o caminho aumentador $s \to v \to u \to t$, como mostra a Figura 26.7(b), então o fluxo tem valor 2. A Figura 26.7(c) mostra a rede residual resultante. Podemos continuar, escolhendo o caminho aumentador $s \to u \to v \to t$ nas iterações ímpares e o caminho aumentador $s \to v \to u \to t$ nas iterações pares. Executaríamos ao todo 2.000.000 aumentos que elevariam o valor do fluxo de apenas uma unidade de cada vez.

O algoritmo de Edmonds-Karp

Podemos melhorar o limite em FORD-FULKERSON encontrando o caminho aumentador p na linha 3 com uma busca em largura. Isto é, escolhemos o caminho aumentador como o caminho *mínimo* de s a t na rede residual, onde cada aresta tem distância (peso) unitária. O método Ford-Fulkerson assim implementado é denominado ***algoritmo de Edmonds e Karp***. Agora, provaremos que o algoritmo Edmonds-Karp é executado no tempo $O(VE_2)$.

A análise depende das distâncias até os vértices na rede residual G_f . O lema a seguir usa a notação $df(u,v)$ para a distância do caminho mínimo de u a v em G_f, onde cada aresta tem distância unitária.

Lema 26.7

Se o algoritmo Edmonds-Karp é executado em uma rede de fluxo $G = (V, E)$ com fonte s e sorvedouro t, então para todos os vértices $v \in V - \{s, t\}$, a distância do caminho mínimo $df\ (s,v)$ na rede residual G_f aumenta monotonicamente com cada aumento de fluxo.

Prova Vamos supor que, para algum vértice $v \in V - \{s, t\}$, existe um aumento de fluxo que provoca diminuição na distância de caminho mínimo de s a v e então deduziremos uma contradição. Seja f o fluxo imediatamente antes do

primeiro aumento que diminui alguma distância de caminho mínimo e seja f' o fluxo imediatamente após. Seja v o vértice com o mínimo df' (s, v), cuja distância foi diminuída pelo aumento, de modo que df' $(s, v) < df(s, v)$. Seja $p = s$ $u \to v$ um caminho mínimo de s a v em $G_{f'}$, de modo que $(u, v) \in E_{f'}$ e

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1. \qquad (26.12)$$

Em razão do modo como escolhemos v, sabemos que a distância do vértice u em relação à fonte s não diminuiu, isto é,

$$\delta_{f'}(s, u) \geq \delta_{f}(s, u). \qquad (26.13)$$

Figura 26.7 (a) Uma rede de fluxo para a qual FORD-FULKERSON pode levar tempo $(E\,|f^*|)$, onde f* é um fluxo máximo, mostrado aqui com $|f^*|$ = 2.000.000. O caminho sombreado é um caminho aumentador com capacidade residual 1. (b) A rede residual resultante, com outro caminho aumentador cuja capacidade residual é 1. (c) A rede residual resultante.

Afirmamos que $(u, v) \notin E_f$. Por quê? Se tivéssemos $(u, v) \in E_f$, então teríamos também

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 && \text{(pelo Lema 24.10, desigualdade triangular)} \\ &\leq \delta_{f'}(s, u) + 1 && \text{(pela desigualdade (26.13))} \\ &= \delta_{f'}(s, v) && \text{(pela equação (26.12))}\ , \end{aligned}$$

o que contradiz nossa hipótese que $df'(s, v) < df'$ (s, v).

Como podemos ter $(u, v) \notin E_f$ e $(u, v) \in E_{f'}$? O aumento deve ter aumentado o fluxo de v para u. O algoritmo Edmonds-Karp sempre aumenta o fluxo ao longo de caminhos mínimos e, portanto, aumentou ao longo de um caminho mínimo de s a u em G_f que tem (v, u) como sua última aresta. Assim,

$$\begin{aligned} \delta_f(s, v) &= \delta_{f'}(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 && \text{(pela desigualdade (26.13))} \\ &= \delta_{f'}(s, u) - 2 && \text{(pela equação (26.12))}\ , \end{aligned}$$

o que contradiz nossa hipótese de que df' $(s, v) < df(s, v)$. Concluímos que a hipótese de que tal vértice v existe é incorreta.

O próximo teorema limita o número de iterações do algoritmo de Edmonds-Karp.

Teorema 26.8

Se o algoritmo Edmonds-Karp é executado em uma rede de fluxo $G = (V, E)$ com fonte s e sorvedouro t, então o número total de aumentos de fluxo executados pelo algoritmo é no máximo $O(V\,E)$.

Prova Dizemos que uma aresta (u, v) em uma rede residual G_f é ***crítica*** em um caminho aumentador p se a capacidade residual de p é a capacidade residual de (u, v), isto é, se $c_f(p) = c_f(u, v)$. Depois de aumentarmos o fluxo ao longo de

um caminho aumentador, qualquer aresta crítica no caminho desaparece da rede residual. Além disso, no mínimo uma aresta em qualquer caminho aumentador deve ser crítica. Mostraremos que cada uma das $|E|$ arestas pode se tornar crítica no máximo $|V|/2$ vezes.

Sejam u e v vértices em V que estão conectados por uma aresta em E. Visto que caminhos aumentadores são caminhos mínimos, quando (u, v) é crítica pela primeira vez, temos

$$\delta_f(s, v) = \delta_f(s, u) + 1.$$

Uma vez aumentado o fluxo, a aresta (u, v) desaparece da rede residual. Ela só pode reaparecer mais tarde em um outro caminho aumentador depois que o fluxo de u a v for diminuído, o que ocorre somente se (v, u) aparecer em um caminho aumentador. Se f' é o fluxo em G quando esse evento ocorre, então temos

$$\delta_{f'}(s, u) = \delta_f(s, v) + 1.$$

Visto que $\delta_{f'}(s, v) \leq \delta_f{}'(s, v)$ pelo Lema 26.7, temos

$$\begin{aligned}\delta_f{}'(s, u) &= \delta_{f'}(s, v) + 1 \\ &\leq \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2 .\end{aligned}$$

Consequentemente, a partir do momento em que (u, v) se torna crítica até o momento em que ela se torna crítica outra vez, a distância de u em relação à fonte aumenta de no mínimo 2. A distância de u em relação à fonte é inicialmente no mínimo 0. Os vértices intermediários em um caminho mínimo de s a u não podem conter s, u ou t (visto que (u, v) em um caminho aumentador implica que $u \neq t$). Então, até u não poder mais ser alcançado da fonte, se é que isso ocorre, sua distância é no máximo $|V| - 2$. Portanto, depois da primeira vez que (u, v) se torna crítica, ela só pode tornar-se crítica novamente no máximo mais $(|V| - 2)/2 = |V|/2 - 1$ vezes para um total de no máximo $|V| / 2$ vezes. Visto que em uma rede residual há $O(E)$ pares de vértices que podem ter uma aresta entre eles, o número total de arestas críticas durante toda a execução do algoritmo Edmonds-Karp é $O(V E)$. Cada caminho aumentador tem no mínimo uma aresta crítica e, consequentemente, o teorema decorre.

Como podemos implementar cada iteração de FORD-FULKERSON no tempo $O(E)$ quando encontramos o caminho aumentador por busca em largura, o tempo de execução total do algoritmo de Edmonds-Karp é $O(VE_2)$. Veremos que os algoritmos push-relabel podem produzir limites ainda melhores. O algoritmo da Seção 26.4 dá um método para conseguir um tempo de execução $O(V_2 E)$, que forma a base para o algoritmo de tempo $O(V_3)$ da Seção 26.5.

Exercícios

26.2-1 Prove que os somatórios na equação (26.6) são iguais aos somatórios na equação (26.7).

26.2-2 Na Figura 26.1(b), qual é o fluxo pelo corte $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? Qual é a capacidade desse corte?

26.2-3 Mostre a execução do algoritmo Edmonds-Karp na rede de fluxo da Figura 26.1(a).

26.2-4 No exemplo da Figura 26.6, qual é o corte mínimo correspondente ao fluxo máximo mostrado? Dos caminhos aumentadores que aparecem no exemplo, qual é o que cancela o fluxo?

26.2-5 Lembre-se de que a construção na Seção 26.1 que converte uma rede de fluxo com várias fontes e vários sorvedouros em uma rede com fonte única e sorvedouro único adiciona arestas com capacidade infinita. Prove que qualquer fluxo na rede resultante tem um valor finito se as arestas da rede original com várias fontes e vários sorvedouros têm capacidade finita.

26.2-6 Suponha que cada fonte s_i em uma rede de fluxo com várias fontes e vários sorvedouros produza exatamente p_i unidades de fluxo, de modo que $\sum_{v \in V} f(s_i, v) = p_i$. Suponha também que cada sorvedouro t_j consuma exatamente q_j unidades, de modo que $\sum_{v \in V} f(v, t_j) = q_j$, onde $\sum_i p_i = \sum_j q_j$. Mostre como converter o problema de encontrar um fluxo f que obedeça a essas restrições adicionais no problema de encontrar um fluxo máximo em uma rede de fluxo com fonte única e sorvedouro único.

26.2-7 Prove o Lema 26.2.

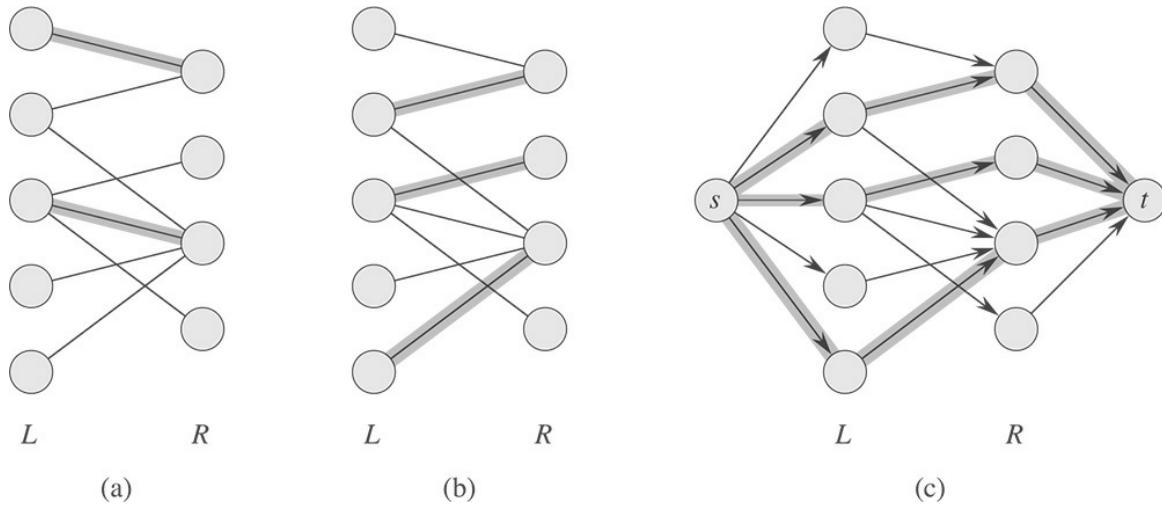
26.2-8 Suponha que redefinimos a rede residual para desautorizar arestas que entrem em s. Prove que o procedimento FORD-FULKERSON ainda calcula corretamente um fluxo máximo.

26.2-9 Suponha que f e f' sejam fluxos em uma rede G e que calculamos o fluxo $f \uparrow f'$. O fluxo aumentado satisfaz a propriedade de conservação de fluxo? Satisfaz a restrição de capacidade?

26.2-10 Mostre como encontrar um fluxo máximo em uma rede $G = (V, E)$ por uma sequência de no máximo $|E|$ caminhos aumentadores. (*Sugestão*: Determine os caminhos *depois* de determinar o fluxo máximo.)

26.2-11 A ***conectividade de aresta*** de um grafo não dirigido é o número mínimo k de arestas que devem ser removidas para desconectar o grafo. Por exemplo, a conectividade de aresta de uma árvore é 1, e a conectividade de aresta de uma cadeia cíclica de vértices é 2. Mostre como determinar a conectividade de aresta de um grafo não dirigido $G = (V, E)$ executando um algoritmo de fluxo máximo em, no máximo, $|V|$ redes de fluxo, cada uma com $O(V)$ vértices e $O(E)$ arestas.

26.2-12 Suponha que temos uma rede de fluxo G cujas capacidades são todas inteiras, e G tenha arestas que entram na fonte s. Seja f um fluxo em G no qual uma das arestas (v, s) que entram na fonte tem $f(v, s) = 1$. Prove que deve existir um outro fluxo f' com $f'(v, s) = 0$ tal que $|f| = |f'|$. Dê um algoritmo de tempo $O(E)$ para calcular f', dado f.

26.2-13 Suponha que você deseje encontrar, entre todos os cortes mínimos em uma rede de fluxo G, um que contenha o menor número de arestas. Mostre como modificar as capacidades de G para criar uma nova rede de fluxo G' na qual qualquer corte mínimo em G' seja um corte com o menor número de arestas em G.

26.3 EMPARELHAMENTO MÁXIMO EM GRAFO BIPARTIDO

Alguns problemas combinatórios podem ser facilmente expressos como problemas de fluxo máximo. O problema de fluxo máximo de várias fontes e vários sorvedouros da Seção 26.1 nos deu um exemplo. Alguns outros problemas combinatórios aparentemente têm pouco a ver com redes de fluxo, mas, na verdade, podem ser reduzidos a problemas de fluxo máximo. Esta seção apresenta um desses problemas: encontrar um emparelhamento máximo em um grafo bipartido. Para resolver esse problema, aproveitaremos uma propriedade de integralidade proporcionada pelo método de Ford-Fulkerson. Também veremos como usar o método de Ford-Fulkerson para resolver o problema de emparelhamento máximo em grafo bipartido em um grafo $G = (V, E)$ no tempo $O(V E)$.

O problema de emparelhamento máximo em grafo bipartido

Dado um grafo não dirigido $G = (V, E)$, um ***emparelhamento*** é um subconjunto de arestas $M \subseteq E$ tal que, para todos os vértices $v \in V$, no máximo uma aresta de M é incidente em v. Dizemos que um vértice $v \in V$ é ***emparelhado*** pelo emparelhamento M se alguma aresta em M é incidente em v; caso contrário, v é ***não***

emparelhado. Um ***emparelhamento máximo*** é um emparelhamento de cardinalidade máxima, isto é, um emparelhamento M tal que, para qualquer emparelhamento M', temos $|M| \geq |M'|$. Nesta seção, restringiremos nossa atenção a determinar emparelhamentos máximos em grafos bipartidos: grafos nos quais o conjunto de vértices pode ser particionado em $V = L \cup R$, onde L e R são disjuntos e todas as arestas em E passam entre L e R. Suporemos ainda que todo vértice em V tem no mínimo uma aresta incidente. A Figura 26.8 ilustra a noção de emparelhamento em um grafo bipartido.

Figura 26.8 Um grafo bipartido $G = (V, E)$ com partição de vértice $V = L \cup R$. **(a)** Um emparelhamento com cardinalidade 2, indicado por arestas sombreadas. **(b)** Um emparelhamento máximo com cardinalidade 3. **(c)** Rede de fluxo correspondente G' **mostrando um fluxo máximo**. Cada aresta tem capacidade unitária. Arestas sombreadas têm fluxo de 1, e as outras arestas não transportam nenhum fluxo. As arestas sombreadas de L a R correspondem às do emparelhamento máximo em (b).

O problema de encontrar um emparelhamento máximo em um grafo bipartido tem muitas aplicações práticas. Como exemplo, poderíamos considerar o emparelhamento de um conjunto L de máquinas com um conjunto R de tarefas que devem ser executadas simultaneamente. Consideramos que a presença da aresta (u, v) em E significa que uma determinada máquina $u \in L$ é capaz de executar uma dada tarefa $v \in R$. Um emparelhamento máximo atribui trabalho ao maior número de máquinas possível.

Encontrando emparelhamento máximo em grafo bipartido

Podemos usar o método Ford-Fulkerson para encontrar emparelhamento máximo em um grafo bipartido não dirigido $G = (V, E)$ em tempo polinomial em $|V|$ e $|E|$. O truque é construir uma rede de fluxo na qual os fluxos correspondem a emparelhamentos, como mostra a Figura 26.8. Definimos a ***rede de fluxo correspondente*** $G' = (V', E')$ para o grafo bipartido G da seguinte maneira: sejam a fonte s e o sorvedouro t novos vértices não pertencentes a V, e seja $V' = V \cup \{s, t\}$. Se a partição de vértices de G é $V = L \cup R$, as arestas dirigidas de G' são as arestas de E, dirigidas de L para R, juntamente com $|V|$ novas arestas dirigidas:

$$E' = \{(s, u) : u \in \mathrm{L}\} \cup \{(u,v) : (u,v) \in E\} \cup \{(v,t) : v \in R\}.$$

Para concluir a construção, atribuímos capacidade unitária a cada aresta em E'. Visto que cada vértice em V tem no mínimo uma aresta incidente, $|E| \geq |V|/2$. Assim, $|E| \leq |E'| = |E| + |V| \leq 3|E|$ e, então, $|E'| = \Theta(E)$.

O lema a seguir mostra que um emparelhamento em G corresponde diretamente a um fluxo na rede de fluxo G' correspondente em G. Dizemos que um fluxo f em uma rede de fluxo $G = (V, E)$ é de ***valor inteiro*** se $f(u, v)$ é um inteiro para todo $(u, v) \in V \times V$.

Lema 26.9

Seja $G = (V, E)$ um grafo bipartido com partição de vértices $V = L \cup R$, e seja $G' = (V', E')$ sua rede de fluxo correspondente. Se M é um emparelhamento em G, então existe um fluxo de valor inteiro f em G' com valor $|f| = |M|$. Ao contrário, se f é um fluxo de valor inteiro em G', então existe um emparelhamento M em G com cardinalidade $|M| = |f|$.

Prova Primeiro mostramos que um emparelhamento M em G corresponde a um fluxo f de valor inteiro em G'. Defina f da seguinte maneira: se $(u, v) \in M$, então $f(s, u) = f(u, v) = f(v, t) = 1$. Para todas as outras arestas $f(u, v) = 0$, definimos $f(u, v) = 0$. É simples verificar que f satisfaz a restrição de capacidade e conservação de fluxo.

Intuitivamente, cada aresta $(u, v) \in M$ corresponde a uma unidade de fluxo em G' que percorre o caminho $s \to u \to v \to t$. Além disso, os caminhos induzidos por arestas em M são de vértices disjuntos, exceto para s e t. O fluxo líquido pelo corte $(L \cup \{s\}, R \cup \{t\})$ é igual a $|M|$; assim, pelo Lema 26.4, o valor do fluxo é $|f| = |M|$.

Para provar o inverso, seja f um fluxo de valor inteiro em G' e seja

$$M = \{(u, v) : u \in L, v \in R, e\, f(u, v) > 0\} .$$

Cada vértice $u \in L$ tem somente uma aresta de entrada, isto é, (s, u), e sua capacidade é 1. Assim, em cada $u \in L$ entra no máximo uma unidade de fluxo positivo e, se uma unidade de fluxo positivo entra, pela conservação de fluxo uma unidade de fluxo positivo deve sair. Além disso, visto que f tem valor inteiro, para cada $u \in L$, a única unidade de fluxo pode entrar em no máximo uma aresta e pode sair no máximo de uma aresta. Assim, uma unidade de fluxo positivo entra em u se e somente se existe exatamente um vértice $v \in R$ tal que $f(u, v) = 1$, e no máximo uma aresta que sai de cada $u \in L$ transporta fluxo positivo. Um argumento simétrico aplica-se a cada $v \in R$. Portanto, o conjunto M é um emparelhamento.

Para verificar que$|M|=|f|$, observe que para todo vértice correspondente $u \in L$ temos $f(s, u)= 1$ e, para toda aresta $(u, v) \in E - M$, temos $f(u, v) = 0$. Consequentemente, $f(L \cup \{s\}, R \cup \{t\})$, o fluxo líquido pelo corte $(L \cup \{s\}, R \cup \{t\})$, é igual a |M|. Aplicando o Lema 26.4, temos que >$|f|= f(L \cup \{s\}, R \cup \{t\}) = |M|$.

Com base no Lema 26.9, gostaríamos de concluir que um emparelhamento máximo em um grafo bipartido G corresponde a um fluxo máximo em sua rede de fluxo correspondente G', e portanto podemos calcular um emparelhamento máximo em G executando um algoritmo de fluxo máximo em G'. O único senão nesse raciocínio é que o algoritmo de fluxo máximo poderia retornar um fluxo em G' para o qual algum $f(u, v)$ não é um inteiro, ainda que o valor de fluxo $|f|$ tenha de ser um inteiro. O teorema a seguir mostra que, se usarmos o método de Ford-Fulkerson, essa dificuldade não pode surgir.

Teorema 26.10 (Teorema de integralidade)

Se a função capacidade c adota somente valores inteiros, então o fluxo máximo f produzido pelo método de Ford-Fulkerson tem a seguinte propriedade: $|f|$ é um inteiro. Além disso, para todos os vértices u e v, o valor de $f(u, v)$ é um inteiro.

Prova A prova é por indução em relação ao número de iterações. Vamos deixá-la para o Exercício 26.3–2.

Agora podemos provar o seguinte corolário para o Lema 26.9:

Corolário 26.11

A cardinalidade de um emparelhamento máximo M em um grafo bipartido G é igual ao valor de um fluxo máximo f em sua rede de fluxo correspondente G'.

Prova Usamos a nomenclatura do Lema 26.9. Suponha que M seja um emparelhamento máximo em G e que o fluxo correspondente f em G' não seja máximo. Então existe um fluxo máximo f' em G' tal que $|f'| > |f|$. Visto que as capacidades em G' são valores inteiros, pelo Teorema 26.10 podemos supor que f' tem valor inteiro. Assim, f' corresponde a um emparelhamento M' em G cardinalidade $|M'| = |f'| > |f| = |M|$, o que contradiz nossa hipótese de que M seja um emparelhamento máximo. De modo semelhante, podemos mostrar que, se f é um fluxo máximo em G', seu emparelhamento correspondente será um emparelhamento máximo em G.

Portanto, dado um grafo bipartido não dirigido G, podemos encontrar um emparelhamento máximo criando a rede de fluxo G', executando o método de Ford-Fulkerson e obtendo diretamente um emparelhamento máximo M pelo fluxo máximo de valor inteiro f encontrado. Visto que qualquer emparelhamento em um grafo bipartido tem cardinalidade no máximo min $(L, R) = O(V)$, o valor do fluxo máximo em G' é $O(V)$. Portanto, podemos encontrar um emparelhamento máximo em um grafo bipartido no tempo $O(V E') = O(V E)$, visto que $|E'| = Q(E)$.

Exercícios

26.3-1 Execute o algoritmo de FORD-FULKERSON na rede de fluxo na Figura 26.8(c) e mostre a rede residual após cada aumento de fluxo. Numere os vértices em L de cima para baixo, de 1 a 5, e em R de cima para baixo de 6 a 9. Para cada iteração, escolha o caminho aumentador que seja lexicograficamente menor.

26.3-2 Prove o Teorema 26.10.

26.3-3 Seja $G = (V, E)$ um grafo bipartido com partição de vértice $V = L \cup R$, e seja G' sua rede de fluxo correspondente. Dê um bom limite superior para o comprimento de qualquer caminho aumentador encontrado em G' durante a execução de FORD-FULKERSON.

26.3-4 ★ Um ***emparelhamento perfeito*** é um emparelhamento no qual todo vértice é emparelhado. Seja $G = (V, E)$ um grafo bipartido não dirigido com partição de vértice $V = L \cup R$, onde $|L| = |R|$. Para qualquer $X \subseteq V$, defina a ***vizinhança*** de X como

$$N(X) = \{y \in V : (x, y) \in E \text{ para algum } x \in X\},$$

isto é, o conjunto de vértices adjacentes a algum membro de X. Prove o ***teorema de Hall***: existe um emparelhamento perfeito em G se e somente se $|A| \leq |N(A)|$ para todo subconjunto $A \subseteq L$.

26.3-5 ★ Dizemos que um grafo bipartido $G = (V, E)$, onde $V = L \cup R$ é ***d–regular*** se todo vértice $v \in V$ tem grau exatamente d. Todo grafo bipartido d-regular tem $|L| = |R|$. Prove que todo grafo bipartido d-regular tem um emparelhamento de cardinalidade $|L|$ demonstrando que um corte mínimo da rede de fluxo correspondente tem capacidade $|L|$.

26.4 ★ ALGORITMOS PUSH-RELABEL

Nesta seção, apresentamos a abordagem "push-relabel" (empurrar-remarcar) para calcular fluxos máximos. Até o momento, muitos dos algoritmos de fluxo máximo assintoticamente mais rápidos são algoritmos push-relabel, e as mais rápidas implementações reais de algoritmos de fluxo máximo se baseiam no método push-relabel. Os métodos push-relabel resolvem também eficientemente outros problemas de fluxo, como o problema do fluxo de custo mínimo. Esta seção apresenta o algoritmo de fluxo máximo "genérico" de Goldberg, que tem uma implementação simples que é executada no tempo $O(V_2 \text{E})$, melhorando assim o limite de $O(VE_2)$ do algoritmo Edmonds-Karp. A Seção 26.5 refina o algoritmo genérico para obter um outro algoritmo push-relabel que é executado no tempo $O(V_3)$.

Algoritmos push-relabel funcionam de uma maneira mais localizada que o método Ford-Fulkerson. Em vez de examinar toda a rede residual para encontrar um caminho aumentador, algoritmos push-relabel agem em um vértice por vez, examinando somente os vizinhos do vértice na rede residual. Além disso, diferentemente do método de Ford-Fulkerson, os algoritmos push-relabel não mantêm a propriedade de conservação de fluxo durante toda a sua execução. Entretanto, eles mantêm um ***pré-fluxo***, que é uma função $f : V \times V \to$ que satisfaz a restrição de capacidade e o seguinte relaxamento da conservação de fluxo:

$$\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) \geq 0$$

para todos os vértices $u \in V - \{s\}$. Isto é, o fluxo que entra em um vértice pode exceder o fluxo que sai. Denominamos essa quantidade

$$e(u) = \sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) \qquad (26.14)$$

excesso de fluxo no vértice u. O excesso em um vértice é a quantidade que representa a diferença entre o fluxo que entra e o fluxo que sai. Dizemos que um vértice $u \in V - \{s, t\}$ está ***transbordando*** se $e(u) > 0$.

Iniciaremos esta seção descrevendo a intuição que fundamenta o método push-relabel. Então investigaremos as duas operações empregadas pelo método: "empurrar" pré-fluxo e "remarcar" um vértice. Finalmente, apresentaremos um algoritmo push-relabel genérico e analisaremos sua correção e seu tempo de execução.

Intuição

Podemos entender a intuição que fundamenta o método push-relabel em termos de fluxos de fluidos: consideramos uma rede de fluxo $G = (V, E)$ como um sistema de tubos interconectados de capacidades dadas. Aplicando essa analogia ao método Ford-Fulkerson, podemos dizer que cada caminho aumentador na rede dá origem a uma corrente adicional de fluido, sem nenhum ponto de derivação, que flui da fonte ao sorvedouro. O método Ford-Fulkerson adiciona iterativamente mais correntes de fluxo até que nada mais seja possível adicionar.

A intuição do algoritmo push-relabel genérico é bastante diferente. Como antes, arestas dirigidas correspondem a tubos. Vértices, que são junções de tubos, têm duas propriedades interessantes. A primeira é que, para acomodar excesso de fluxo, cada vértice tem um tubo de saída (uma derivação), que conduz esse excesso até um reservatório arbitrariamente grande que pode acumular fluido. A segunda é que cada vértice, seu reservatório e todas as suas conexões tubulares estão em uma plataforma cuja altura aumenta à medida que o algoritmo progride.

As alturas dos vértices determinam como o fluxo é empurrado: empurramos o fluxo em declive, isto é, de um vértice mais alto para um vértice mais baixo. O fluxo de um vértice mais baixo para um vértice mais alto pode ser positivo, mas as operações que empurram o fluxo só o empurram em declive. Fixamos a altura da fonte em $|V|$ e a altura do sorvedouro em 0. As alturas de todos os outros vértices começam em 0 e aumentam com o tempo. Primeiro, o algoritmo envia o máximo de fluxo possível em declive da fonte ao sorvedouro. A quantidade enviada é exatamente a suficiente para encher cada tubo que sai da fonte até sua capacidade nominal, isto é, o algoritmo envia a capacidade do corte $(s, V - \{s\})$. Quando entra pela primeira vez em um vértice intermediário, o fluxo é coletado no reservatório do vértice de onde, em determinado momento, ele é empurrado em declive.

A certa altura, podemos descobrir que os únicos tubos que saem de um vértice u e ainda não estão saturados de fluido se conectam com vértices que estão no mesmo nível que u ou que estão acima de u. Nesse caso, para livrar do excesso de fluxo um vértice u que está transbordando, temos de aumentar sua altura — uma operação denominada "remarcar" o vértice u. Aumentamos a altura desse vértice de uma unidade a mais que a altura do mais baixo de seus vizinhos ao qual está conectado por um tubo não saturado. Portanto, depois que um vértice é remarcado (*relabeled*) ele tem, no mínimo, um tubo de saída pelo qual podemos empurrar mais fluxo.

A certa altura, todo o fluxo que poderia chegar até o sorvedouro já chegou. Nada mais pode chegar porque os tubos obedecem a restrições de capacidade; a quantidade de fluxo que passa por qualquer corte ainda é limitada pela capacidade do corte. Então, para que o pré-fluxo se torne um fluxo válido, o algoritmo envia de volta à fonte o excesso coletado nos reservatórios de vértices que estavam transbordando, continuando a remarcar vértices com uma altura maior que a altura fixa $|V|$ da fonte. Como veremos, tão logo tenhamos esvaziado todos os reservatórios, o pré-fluxo não é somente um fluxo "válido"; é também um fluxo máximo.

As operações básicas

Pela discussão precedente, vemos que o algoritmo push-relabel executa duas operações básicas: empurra o excesso de fluxo de um vértice até um de seus vizinhos e remarca um vértice. As situações às quais essas operações se aplicam dependem das alturas dos vértices, que agora definimos exatamente.

Seja $G = (V, E)$ uma rede de fluxo com fonte s e sorvedouro t, e seja f um pré-fluxo em G. Uma função $h : V \rightarrow$ é uma ***função altura***[3] se $h(s) = |V|$, $h(t) = 0$ e

$$h(u) \leq h(v) + 1$$

para toda aresta residual $(u, v) \in E_f$. Obtemos imediatamente o lema a seguir.

Lema 26.12

Seja $G = (V, E)$ uma rede de fluxo, seja f um pré-fluxo em G e seja h uma função altura em V. Para quaisquer dois vértices $u, v \in V$, se $h(u) > h(v) + 1$, então (u, v) não é uma aresta no grafo residual.

A operação empurrar

A operação básica PUSH(u, v) se aplica se u é um vértice que está transbordando, $c_f(u, v) > 0$ e $h(u) = h(v) + 1$. O pseudocódigo que apresentamos em seguida atualiza o pré-fluxo f e os excessos de fluxo para u e v. Ele supõe que podemos calcular a capacidade residual $c_f(u, v)$ em tempo constante dados c e f. Mantemos o excesso de fluxo armazenado em um vértice u como o atributo $u.e$ e a altura de u como o atributo $u.h$. A expressão $d_f(u, v)$ é uma variável temporária que armazena a quantidade de fluxo que pode ser empurrada de u para v.

PUSH(u,v)

1 // **Aplica-se quando:** u está transbordando, $c_f(u,v) > 0$, e $u.h = v.h + 1$.
2 // **Ação:** Empurrar $\Delta_f(u,v) = \min(u.e, c_f(u,v))$ unidades de fluxo de u até v.
3 $\Delta_f(u,v) = \min(u.e, c_f(u,v))$
4 **if** $(u,v) \in E$
5 $\quad (u,v).f = (u,v).f + \Delta_f(u,v)$
6 **else** $(u,v).f = (u,v).f - \Delta_f(u,v)$
7 $u.e = u.e - \Delta_f(u,v)$
8 $v.e = v.e + \Delta_f(u,v)$

O código para PUSH funciona da seguinte maneira: como o vértice u tem um excesso positivo $u.e$ e a capacidade residual de (u, v) é positiva, podemos aumentar o fluxo de u a v de D$f(u, v) = \min(u.e, c_f(u,v))$ sem que $u.e$ torne-se negativo nem que a capacidade $c(u, v)$ seja excedida. A linha 3 calcula o valor D$f(u, v)$ e as linhas 4–6 atualizam f. A linha 5 aumenta o fluxo na aresta (u, v) porque estamos empurrando fluxo por uma aresta residual que também é uma aresta original. A linha 6 diminui o fluxo na aresta (v, u) porque, na verdade, a aresta residual é o inverso de uma aresta

na rede original. Por fim, as linhas 7–8 atualizam os excessos de fluxo que entram nos vértices u e v. Assim, se f é um pré-fluxo antes de PUSH ser chamada, continua sendo um pré-fluxo depois.

Observe que nada no código de PUSH depende das alturas de u e v; ainda assim proibimos que ele seja invocado, a menos que $u.h = v.h + 1$. Portanto, empurramos o excesso de fluxo para baixo por uma altura diferencial de apenas 1. Pelo Lema 26.12, não existem arestas residuais entre dois vértices cujas diferenças entre alturas sejam maiores do que 1 e, assim, visto que o atributo h é de fato uma função altura não teríamos nada a ganhar se permitirmos que o fluxo seja empurrado para baixo por um diferencial de altura maior que 1.

Dizemos que PUSH(u, v) é um ***empurrão*** de u a v. Se uma operação push se aplicar a alguma aresta (u, v) que sai de um vértice u, dizemos também que tal operação se aplica a u. Ela é um ***empurrão saturador*** se a aresta (u, v) na rede residual tornar-se ***saturada*** ($c_f(u, v) = 0$ depois); caso contrário, ela é um ***empurrão não saturador***. Se uma aresta torna-se saturada, ela desaparece da rede residual. Um lema simples caracteriza uma consequência de um empurrão não saturador.

Lema 26.13

Após um empurrão não saturador de u a v, o vértice u não está mais transbordando.

Prova Visto que o empurrão foi não saturador, a quantidade de fluxo D*f*(u, v) realmente empurrada tem de ser igual a $u.e$ antes do empurrão. Como $u.e$ é reduzido dessa quantidade, ele se torna 0 após o empurrão.

A operação remarcar

A operação básica RELABEL(u) se aplica se u está transbordando e se $u.h \leq v.h$ para todas as arestas $(u, v) \in E_f$. Em outras palavras, podemos remarcar um vértice u que está transbordando se, para todo vértice v para o qual há capacidade residual de u a v, o fluxo não puder ser empurrado de u para v porque v não está abaixo de u. (Lembre-se de que, por definição, nem a fonte s nem o sorvedouro t podem transbordar, portanto nem s nem t são candidatos à remarcação.)

```
RELABEL(u)
1   // Aplica-se quando: u está transbordando e, para todo v ∈ V tal que (u, v) ∈ E_f,
              temos u.h ≤ v.h.
2   // Ação: Aumentar a altura de u.
3   u.h = 1 + min{v.h : (u, v) ∈ E_f}
```

Quando chamamos a operação RELABEL(u), dizemos que o vértice u é ***remarcado*** (*relabeled*). Observe que, quando u é remarcado, E_f deve conter no mínimo uma aresta que saia de u, de modo que a minimização no código é em relação a um conjunto não vazio. Essa propriedade decorre da hipótese de que u está transbordando, o que, por sua vez, nos diz que

$$u.e = \sum_{v \in V} f(v,u) - \sum_{v \in V} f(v,u) > 0$$

Visto que todos os fluxos são não negativos, então temos de ter no mínimo um vértice v tal que $(v,u).f > 0$. Mas, então, $c_f(u,v) > 0$, o que implica que $(u,v) \in E_f$. Assim, a operação RELABEL(u) dá a u a maior altura permitida pelas restrições impostas às funções altura.

O algoritmo genérico

O algoritmo push-relabel genérico utiliza a seguinte subrotina para criar um pré-fluxo inicial na rede de fluxo:

```
INITIALIZE-PREFLOW(G, s)

1   for cada vértice v ∈ G.V
2       v.h = 0
3       v.e = 0
4   for cada aresta (u,v) ∈ G.E
5       (u,v).f = 0
6   s.h = |G.V|
7   for cada vértice v ∈ s.Adj
8       (s,v).f = c(s,v)
9       v.e = c(s,v)
10      s.e = s.e − c(s,v)
```

INITIALIZE-PREFLOW cria um pré-fluxo inicial f definido por

$$(u,v).f = \begin{cases} c(u,v) & \text{se } u = s\text{ ,} \\ 0 & \text{caso contrário.} \end{cases} \quad (26.15)$$

Isto é, enchemos cada aresta que sai da fonte s até a capacidade total, e todas as outras arestas não transportam nenhum fluxo. Para cada vértice v adjacente à fonte, temos inicialmente $v.e = c(s, v)$, e inicializamos $s.e$ como o negativo da soma dessas capacidades. O algoritmo genérico também começa com uma função altura inicial h, dada por

$$u.h = \begin{cases} |V| & \text{se } u = s\text{ ,} \\ 0 & \text{caso contrário .} \end{cases} \quad (26.16)$$

A equação (26.16) define uma função altura porque as únicas arestas (u, v) para as quais $u.h > v.h + 1$ são aquelas para as quais $u = s$, e essas arestas são saturadas, o que significa que não estão na rede residual.

A inicialização, seguida por uma sequência de operações empurrão e remarcação, executadas sem qualquer ordem definida, produz o algoritmo GENERIC-PUSH-RELABEL:

```
GENERIC-PUSH-RELABEL(G)

1   INITIALIZE-PREFLOW(G, s)
2   while existir uma operação empurrão ou remarcação aplicável
3       selecionar uma operação de empurrão ou remarcação aplicável e executá-la
```

O lema a seguir nos informa que, desde que exista um vértice que está transbordando, no mínimo uma das duas operações básicas se aplica.

Lema 26.14 (Um vértice que está transbordando pode ser empurrado ou remarcado)

Seja $G = (V, E)$ uma rede de fluxo com fonte s e sorvedouro t, seja f um pré-fluxo e seja h alguma função altura para f. Se u é qualquer vértice que está transbordando, então uma operação empurrão ou remarcação é aplicável.

Prova Para qualquer aresta residual (u, v), temos $h(u) \leq h(v) + 1$ porque h é uma função altura. Se uma operação empurrão não se aplica a um vértice u que está transbordando, então, para todas as arestas residuais (u, v), devemos ter $h(u) < h(v) + 1$, o que implica $h(u) \leq h(v)$. Assim, uma operação remarcação aplica-se a u.

Correção do método push-relabel

Para mostrar que o algoritmo push-relabel genérico resolve o problema de fluxo máximo, em primeiro lugar temos de provar que, se ele terminar, o pré-fluxo f é um fluxo máximo. Mais adiante provaremos que ele termina. Começamos com algumas observações sobre a função altura h.

Lema 26.15 (Alturas de vértices nunca diminuem)

Durante a execução do procedimento GENERIC-PUSH-RELABEL em uma rede de fluxo $G = (V, E)$, para cada vértice $u \in V$, a altura $u.h$ nunca diminui. Além disso, sempre que uma operação remarcação é aplicada a um vértice u, sua altura $u.h$ aumenta de no mínimo 1.

Prova Como as alturas de vértices só mudam durante operações remarcação, basta provar a segunda afirmação do lema. Se o vértice u está prestes a passar por uma operação remarcação, então, para todos os vértices v tais que $(u, v) \in E_f$, temos $u.h \leq v.h$. Portanto, $u.h < 1 + \min\{v.h : (u, v) \in E_f\}$ e, assim, a operação deve aumentar $u.h$.

Lema 26.16

Seja $G = (V, E)$ uma rede de fluxo com fonte s e sorvedouro t. Então, a execução de GENERIC-PUSH-RELABEL em G mantém o atributo h como uma função altura.

Prova A prova é por indução em relação ao número de operações básicas executadas. Inicialmente, h é uma função altura, como já observamos.

Afirmamos que, se h é uma função altura, então uma operação RELABEL(u) mantém h como uma função altura. Se examinarmos uma aresta residual $(u, v) \in E_f$ que sai de u, então a operação RELABEL(u) assegura que $u.h \leq v.h + 1$ depois dela. Agora, considere uma aresta residual (w, u) que entra em u. Pelo Lema 26.15, $w.h \leq u.h + 1$ antes da operação RELABEL(u) implica $w.h < u.h + 1$ depois dela. Assim, a operação RELABEL(u) mantém h como uma função altura.

Agora, considere uma operação PUSH(u, v). Essa operação pode adicionar a aresta (v, u) a E_f e pode remover (u, v) de E_f. No primeiro caso, temos $v.h = u.h - 1 < u.h + 1$ e, assim, h continua a ser uma função altura. No último caso, remover (u, v) da rede residual remove a restrição correspondente e, novamente, h continua a ser uma função altura.

O lema a seguir dá uma importante propriedade de funções altura.

Lema 26.17

Seja $G = (V, E)$ uma rede de fluxo com fonte s e sorvedouro t, seja f um pré-fluxo em G e seja h uma função altura em V. Então, não existe nenhum caminho da fonte s ao sorvedouro t na rede residual G_f.

Prova Considere, por contradição, que G_f contenha um caminho p de s a t, onde $p = \langle v_0, v_1, ..., v_k \rangle$, $v_0 = s$ e $v_k = t$. Sem perda da generalidade, p é um caminho simples e, assim, $k < |V|$. Para $i = 0, 1, ..., k - 1$, a aresta $(v_i, v_{i+1}) \in E_f$. Como h é uma função altura, $h(v_i) \leq h(v_{i+1}) + 1$ para $i = 0, 1, ..., k - 1$. A combinação dessas desigualdades em relação ao caminho p produz $h(s) \leq h(t) + k$. Mas, como $h(t) = 0$, temos $h(s) \leq k < |V|$, o que contradiz o requisito que $h(s) = |V|$ em uma função altura.

Agora estamos prontos para mostrar que, se o algoritmo push-relabel genérico terminar, o pré-fluxo que ele calcula será um fluxo máximo

Teorema 26.18 (Correção do algoritmo push-relabel genérico)

Se o algoritmo GENERIC-PUSH-RELABEL termina quando executado em uma rede de fluxo $G = (V, E)$ com fonte s e sorvedouro t, então o pré-fluxo f que ele calcula será um fluxo máximo para G.

Prova Usamos o seguinte invariante de laço:

Cada vez que o teste do laço **while** na linha 2 em GENERIC-PUSH-RELABEL é executado, f é um pré-fluxo.

Inicialização: INITIALIZE-PREFLOW faz f um pré-fluxo.

Manutenção: As únicas operações dentro do laço **while** das linhas 2–3 são empurrão e relabel.
As operações remarcação afetam somente atributos de altura e não os valores de fluxo; consequentemente, não afetam se f é ou não um pré-fluxo. Como demonstramos ao explicar PUSH(u, v), se f é um pré-fluxo antes de uma operação empurrão, continua a ser um pré-fluxo depois dela.

Término: No término, cada vértice em V - $\{s, t\}$ deve ter um excesso 0 porque, pelo Lema 26.14 e pelo invariante no qual f é sempre um pré-fluxo, não há nenhum vértice que esteja transbordando. Portanto, f é um fluxo. O Lema 26.16 mostra que h é uma função altura no término e, assim, o Lema 26.17 nos diz que não existe nenhum caminho de s a t na rede residual G_f. Portanto, pelo teorema de fluxo máximo/corte mínimo (Teorema 26.6), f é um fluxo de máximo.

Análise do método push-relabel

Para mostrar que o algoritmo push-relabel genérico de fato termina, limitaremos o número de operações que ele executa. Limitamos separadamente cada um dos três tipos de operações: renomeações, empurrões saturadores e empurrões não saturadores. Conhecendo esses limites, é um problema de solução direta conceber um algoritmo que seja executado no tempo $O(V_2E)$. Contudo, antes de iniciarmos a análise, provaremos um importante lema. Lembre-se de que na rede residual permitimos que arestas entrem na fonte.

Lema 26.19

Seja $G = (V, E)$ uma rede de fluxo com fonte s e sorvedouro t, e seja f um pré-fluxo em G. Então, para qualquer vértice x transbordando, existe um caminho simples de x a s na rede residual G_f.

Prova Para um vértice x transbordando, seja $U = \{v$: existe um caminho simples de x a v em $G_f\}$, e suponha, por contradição, que $s \notin U$. Seja $U = V - U$.

Notando a definição de excesso de fluxo dada pela equação (26.14), somamos sobre todos os vértices em U e observamos que $V = U \cup U$, para obter

$$\begin{aligned}
&\sum_{u \in U} e(u) \\
&\quad = \sum_{u \in U}\left(\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v)\right) \\
&\quad = \sum_{u \in U}\left(\left(\sum_{v \in U} f(v,u) + \sum_{v \in \bar{U}} f(v,u)\right) - \left(\sum_{v \in U} f(u,v) + \sum_{v \in \bar{U}} f(u,v)\right)\right) \\
&\quad = \sum_{u \in U}\sum_{v \in U} f(v,u) + \sum_{u \in U}\sum_{v \in \bar{U}} f(v,u) - \sum_{u \in U}\sum_{v \in U} f(u,v) - \sum_{u \in U}\sum_{v \in \bar{U}} f(u,v) \\
&\quad = \sum_{u \in U}\sum_{v \in \bar{U}} f(v,u) - \sum_{u \in U}\sum_{v \in \bar{U}} f(u,v).
\end{aligned}$$

Sabemos que a quantidade $\sum_{u \in U} e(u)$ deve ser positiva porque $e(x) > 0$, $x \in U$, todos os vértices exceto s têm excesso não negativo e, por hipótese, $s \notin U$. Assim, temos

$$\sum_{u \in U} \sum_{v \in \bar{U}} f(v,u) - \sum_{u \in U} \sum_{v \in \bar{U}} f(v,u) > 0. \tag{26.17}$$

Todos os fluxos de arestas são não negativos e, portanto, para a equação (26.17) ser válida, temos de ter $\sum_{u \in U} \sum_{v \in U} f(v, u) > 0$. Consequentemente, deve existir no mínimo um par de vértices $u' \in U$ e $v' \in U$ com $f(v', u') > 0$. Mas, se $f(v', u') > 0$, deve existir uma aresta residual u', v', o que significa que há um caminho simples de x $u' \to v'$ (o caminho x $u' \to v'$), o que contradiz a definição de U.

O próximo lema limita as alturas dos vértices e seu corolário limita o número de operações relabel executadas no total.

Lema 26.20

Seja $G = (V, E)$ uma rede de fluxo com fonte s e sorvedouro t. Em qualquer instante durante a execução de GENERIC-PUSH-RELABEL em G, temos $u.h \le 2\,|V| - 1$ para todos os vértices $u \in V$.

Prova As alturas da fonte s e do sorvedouro t nunca mudam porque esses vértices, por definição, não transbordam. Assim, sempre temos $s.h = |V|$ e $t.h = 0$, os quais não são maiores que $2|V| - 1$.

Agora considere qualquer vértice $u \in V - \{s, t\}$. Inicialmente, $u.h = 0 \le 2\,|V| - 1$. Mostraremos que, após cada operação relabel, ainda temos $u.h \le 2\,|V| - 1$. Quando u é remarcado, está transbordando e o Lema 26.19 nos diz que existe um caminho simples p de u a s em G_f. Seja $p = \langle v_0, v_1, ..., v_k \rangle$, onde $v_0 = u$, $v_k = s$ e $k \le |V| - 1$, porque p é simples. Para $i = 0, 1, ..., k - 1$, temos $(v_i, v_{i+1}) \in E_f$ e, portanto, pelo Lema 26.16, $v_i.h \le v_{i+1}.h + 1$. Expandir essas desigualdades no caminho p produz $u.h = v_0.h \le v_k.h + k \le s.h + (|V| - 1) = 2|V| - 1$.

Corolário 26.21(Limite em operações remarcação)

Seja $G = (V, E)$ uma rede de fluxo com fonte s e sorvedouro t. Então, durante a execução de GENERIC-PUSH-RELABEL em G, o número de operações remarcação é no máximo $2|V| - 1$ por vértice e no máximo $(2|V| - 1)(|V| - 2) < 2|V|^2$ no total.

Prova Somente os $|V|$ - 2 vértices em V - $\{s, t\}$ podem ser renomeados. Seja $u \in V$ - $\{s, t\}$. A operação RELABEL(u) aumenta $u.h$. O valor de $h[u]$ é inicialmente 0 e, pelo Lema 26.20, cresce até no máximo $2|V| - 1$. Assim, cada vértice $u \in V - \{s, t\}$ é remarcado no máximo $2|V| - 1$ vezes, e o número total de operações remarcação executadas é no máximo $(2|V| - 1)(|V| - 2) < 2|V|^2$.

O Lema 26.20 também nos ajuda a limitar o número de empurrões saturadores.

Lema 26.22 (Limite para empurrões saturadores)

Durante a execução de GENERIC-PUSH-RELABEL em qualquer rede de fluxo $G = (V, E)$, o número de empurrões saturadores é menor que $2|V||E|$.

Prova Para qualquer par de vértices $u, v \in V$, contaremos os empurrões saturadores de u a v e de v a u juntos e os denominaremos empurrões saturadores entre u e v. Se existirem quaisquer desses empurrões, no mínimo um de (u, v) e (v, u) é na realidade uma aresta em E. Agora, suponha que ocorreu um empurrão saturador de u a v. Nesse instante, $v.h = u.h - 1$. Para que mais tarde ocorra um outro empurrão de u a v, em primeiro lugar o algoritmo tem de empurrar

o fluxo de v até u, o que não pode acontecer até que $v.h = u.h + 1$. Visto que $u.h$ nunca diminui, para $v.h = u.h + 1$, o valor de $v.h$ deve aumentar de no mínimo duas unidades. Da mesma forma, $u.h$ deve aumentar de no mínimo dois entre empurrões saturadores de v a u. As alturas começam em 0 e, pelo Lema 26.20, nunca excedem $2|V| - 1$, o que implica que o número de vezes que a altura de qualquer vértice pode aumentar de duas unidades é menor que $|V|$. Visto que no mínimo uma de $u.h$ e $v.h$ deve aumentar de duas unidades entre quaisquer dois pushes saturadores entre u e v, há um número menor que $2|V|$ de empurrões saturadores entre u e v. Multiplicando pelo número de arestas temos um limite menor que $2|V||E|$ para o número total de empurrões saturadores.

O lema a seguir limita o número de empurrões não saturadores no algoritmo push-relabel genérico.

Lema 26.23 (Limite para empurrões não saturadores)

Durante a execução de GENERIC-PUSH-RELABEL em qualquer rede de fluxo $G = (V, E)$, o número de empurrões não saturadores é menor que $4|V|^2 (|V| + |E|)$.

Prova Defina uma função potencial $= \sum_{v:e(v)>0} v.h$. Inicialmente, $= 0$, e o valor de pode mudar após cada operação relabel, empurrão saturador e empurrão não saturador. Limitaremos as quantidades que as operações empurrão saturador e remarcação podem contribuir para o aumento de . Então, mostraremos que cada empurrão não saturador deve diminuir F de no mínimo 1, e usaremos esses limites para deduzir um limite superior para o número de empurrões não saturadores.

Vamos examinar os dois modos possíveis de aumentar . O primeiro é remarcar um vértice u, o que aumenta de menos que $2|V|$, já que o conjunto no qual a soma é efetuada é o mesmo, e relabel não pode aumentar a altura de u para mais do que sua máxima altura possível que, pelo Lema 26.20, é no máximo $2|V| - 1$. O segundo é executar um empurrão saturador de um vértice u até um vértice v, o que aumenta de menos que $2|V|$, já que nenhuma altura muda e só o vértice v, cuja altura é no máximo $2|V| - 1$, poderia se tornar um vértice que está transbordando.

Agora, mostramos que um empurrão não saturador de u para v diminui F de no mínimo 1. Por quê? Antes do empurrão não saturador, u estava transbordando e v poderia ou não estar transbordando. Pelo Lema 26.13, u não está mais transbordando depois do empurrão. Além disso, a menos que v seja a fonte, ele poderia ou não estar transbordando depois do empurrão. Então, a função potencial diminuiu de exatamente $u.h$ e aumentou de 0 ou de $v.h$. Visto que $u.h - v.h = 1$, o efeito líquido é que a função potencial diminuiu de no mínimo 1.

Assim, durante o curso do algoritmo, a quantidade total de aumento em se deve a remarcações e empurrões saturados, e o Corolário 26.21 e o Lema 26.22 restringem o aumento a menos de $(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2 (|V| + |E|)$. Visto que ≥ 0, a quantidade total de diminuição e, portanto, o número total de empurrões não saturadores é menor que $4|V|^2 (|V| + |E|)$.

Agora, que o número de remarcações, empurrões saturadores e empurrões não saturadores foi limitado, definimos o cenário para a análise seguinte do procedimento GENERIC-PUSH-RELABEL e, consequentemente, de qualquer algoritmo baseado no método push-relabel.

Teorema 26.24

Durante a execução de GENERIC-PUSH-RELABEL em qualquer rede de fluxo $G = (V, E)$, o número de operações básicas é $O(V_2E)$.

Prova Imediata pelo Corolário 26.21 e Lemas 26.22 e 26.23.

Assim, o algoritmo termina após $O(V_2E)$ operações. Agora só falta dar um método eficiente para implementar cada operação e escolher uma operação adequada para executar.

Corolário 26.25

Existe uma implementação do algoritmo push-relabel genérico que é executada no tempo $O(V_2E)$ em qualquer rede de fluxo $G = (V, E)$.

Prova O Exercício 26.4–2 pede que você mostre como implementar o algoritmo genérico com uma sobrecarga de $O(V)$ por operação remarcação e $O(1)$ por empurrão. O exercício pede também que você projete uma estrutura de dados que permita escolher uma operação aplicável no tempo $O(1)$. Então o corolário decorre.

Exercícios

26.4-1 Prove que, depois que o procedimento INITIALIZE-PREFLOW(G, s) termina, temos $s.e \leq -|f^*|$, onde f^* é um fluxo máximo para G.

26.4-2 Mostre como implementar o algoritmo push-relabel genérico usando o tempo $O(V)$ por operação remarcação, o tempo $O(1)$ por empurrão e o tempo $O(1)$ para selecionar uma operação aplicável, para um tempo total de $O(V_2E)$.

26.4-3 Prove que o algoritmo push-relabel genérico gasta somente um tempo total de $O(V E)$ na execução de todas as $O(V_2)$ operações remarcação.

26.4-4 Suponha que encontramos um fluxo máximo em uma rede de fluxo $G = (V, E)$ usando um algoritmo push-relabel. Dê um algoritmo rápido para encontrar um corte mínimo em G.

26.4-5 Dê um algoritmo push-relabel eficiente para encontrar um emparelhamento máximo em um grafo bipartido. Analise seu algoritmo.

26.4-6 Suponha que todas as capacidades de arestas em uma rede de fluxo $G = (V, E)$ estejam no conjunto $\{1, 2, ..., k\}$. Analise o tempo de execução do algoritmo push-relabel genérico em termos de $|V|$, $|E|$ e k. (*Sugestão*: Quantas vezes cada aresta pode suportar um empurrão não saturador antes de ficar saturada?)

26.4-7 Mostre que poderíamos mudar a linha 6 de INITIALIZE-PREFLOW para 6 $s.h = |G.V| - 2$ sem afetar a correção ou o desempenho assintótico do algoritmo push-relabel genérico.

26.4-8 Seja $d_f(u, v)$ a distância (número de arestas) de u a v na rede residual G_f. Mostre que o procedimento GENERIC-PUSH-RELABEL mantém as seguintes propriedades: $u.h < |V|$ implica $u.h \leq d_f(u, t)$ e $u.h \geq |V|$ implica $u.h - |V| \leq d_f(u, s)$.

26.4-9 ★Como no exercício anterior, seja $d_f(u, v)$ a distância de u a v na rede residual G_f. Mostre como modificar o algoritmo PUSH-RELABEL genérico para manter as seguintes propriedades: $u.h < |V|$ implica $u.h = d_f(u, t)$, e $u.h \geq |V|$ implica $u.h - |V| = d_f(u, s)$. O tempo total que sua implementação dedica à manutenção dessa propriedade deve ser $O(V E)$.

26.4-10 Mostre que o número de empurrões não saturadores executados pelo procedimento GE- NERIC-PUSH-RELABEL em uma rede de fluxo $G = (V, E)$ é no máximo $4|V|^2|E|$ para $|V| \geq 4$.

26.5 ★ O ALGORITMO RELABEL-TO-FRONT

O método Push-Relabel nos permite aplicar as operações básicas em absolutamente qualquer ordem. Porém, escolhendo a ordem com cuidado e administrando eficientemente a estrutura de dados da rede, podemos resolver o problema de fluxo máximo mais rapidamente que o limite $O(V_2E)$ dado pelo Corolário 26.25. Examinaremos agora o algoritmo relabel-to-front, um algoritmo push-relabel cujo tempo de execução é $O(V_3)$, que é assintoticamente no mínimo tão bom quanto $O(V_2\mathrm{E})$ e até melhor para redes densas.

O algoritmo relabel-to-front mantém uma lista dos vértices da rede. Começando na frente, o algoritmo varre a lista, selecionando repetidamente um vértice u que está transbordando e depois "descarregando-o", isto é, executando operações empurrão e remarcação até u não ter mais um excesso positivo. Sempre que um vértice é remarcado, nós o deslocamos para a frente da lista (daí o nome "relabel-to-front"), e o algoritmo inicia novamente sua varredura.

A correção e a análise do algoritmo relabel-to-front dependem da noção de arestas "admissíveis": as arestas na rede residual pelas quais o fluxo pode ser empurrado. Depois de provar algumas propriedades da rede de arestas admissíveis, investigaremos a operação de descarga, e apresentaremos e analisaremos o algoritmo relabel-to-front propriamente dito.

Arestas e redes admissíveis

Se $G = (V, E)$ é uma rede de fluxo com fonte s e sorvedouro t, f é um pré-fluxo em G e h é uma função altura, dizemos que (u, v) é uma ***aresta admissível*** se $c_f(u, v) > 0$ e $h(u) = h(v) + 1$. Caso contrário, (u, v) é ***inadmissível***. A ***rede admissível*** é $G_{f}{}^{h} = (V, E_{f}{}^{h})$, onde $E_{f}{}^{h}$ é o conjunto de arestas admissíveis.

A rede admissível consiste nas arestas pelas quais podemos empurrar o fluxo. O lema a seguir mostra que essa rede é um grafo acíclico dirigido (gad).

Lema 26.26 (A rede admissível é acíclica)

Se $G = (V, E)$ é uma rede de fluxo, f é um pré-fluxo em G e h é uma função altura em G, então a rede admissível $G_{f}{}^{h} = (V, E_{f}{}^{h})$ é acíclica.

Prova A prova é por contradição. Suponha que $G_{f}{}^{h}$ contenha um ciclo $p = \langle v_0, v_1, ..., v_k\rangle$, onde $v_0 = v_k$ e $k > 0$. Visto que cada aresta em p é admissível, temos $h(v_{i - 1}) = h(v_i) + 1$ para $i = 1, 2, ..., k$. O somatório em torno do ciclo dá

$$\sum_{i=1}^{k} h(v_{i-1}) = \sum_{i=1}^{k} (h(v_i) + 1)$$
$$= \sum_{i=1}^{k} h(v_i) + k$$

Como cada vértice no ciclo p aparece uma vez em cada um dos somatórios, deduzimos a contradição de que $0 = k$.

Os dois lemas seguintes mostram como as operações empurrão e remarcação mudam a rede admissível.

Lema 26.27

Seja $G = (V, E)$ uma rede de fluxo, seja f um pré-fluxo em G e suponha que o atributo h seja uma função altura. Se um vértice u está transbordando e (u, v) é uma aresta admissível, então Push(u, v) se aplica. A operação não cria nenhuma nova aresta admissível, mas pode transformar (u, v) em inadmissível.

Prova Pela definição de aresta admissível, podemos empurrar fluxo de u a v. Visto que u está transbordando, a operação Push(u, v) se aplica. A única aresta residual nova que a operação de empurrar fluxo de u a v pode criar é $(v,$

u). Como $h(v) = h(u) - 1$, a aresta (v, u) não pode se tornar admissível. Se a operação é um empurrão saturador, então $c_f(u, v) = 0$ daí em diante e (u, se torna inadmissível.

Lema 26.28

Seja $G = (V, E)$ uma rede de fluxo, seja f um pré-fluxo em G e suponha que o atributo h seja uma função altura. Se um vértice u está transbordando e não há nenhuma aresta admissível saindo de u, então RELABEL(u) se aplica. Após a operação remarcação, existe no mínimo uma aresta admissível saindo de u, mas não existe nenhuma aresta admissível entrando em u.

Prova Se u está transbordando, pelo Lema 26.14, uma operação empurrão ou uma operação remarcação se aplica a esse vértice. Se não existe nenhuma aresta admissível saindo de u, então nenhum fluxo pode ser empurrado de u e, portanto, RELABEL(u) se aplica. Depois da operação remarcação, $u.h = 1 + \min \{v.h : (u, v) \in E_f\}$. Assim, se v é um vértice que realiza o mínimo nesse conjunto, a aresta (u, v) se torna admissível. Consequentemente, após a operação relabel existe no mínimo uma aresta admissível saindo de u.

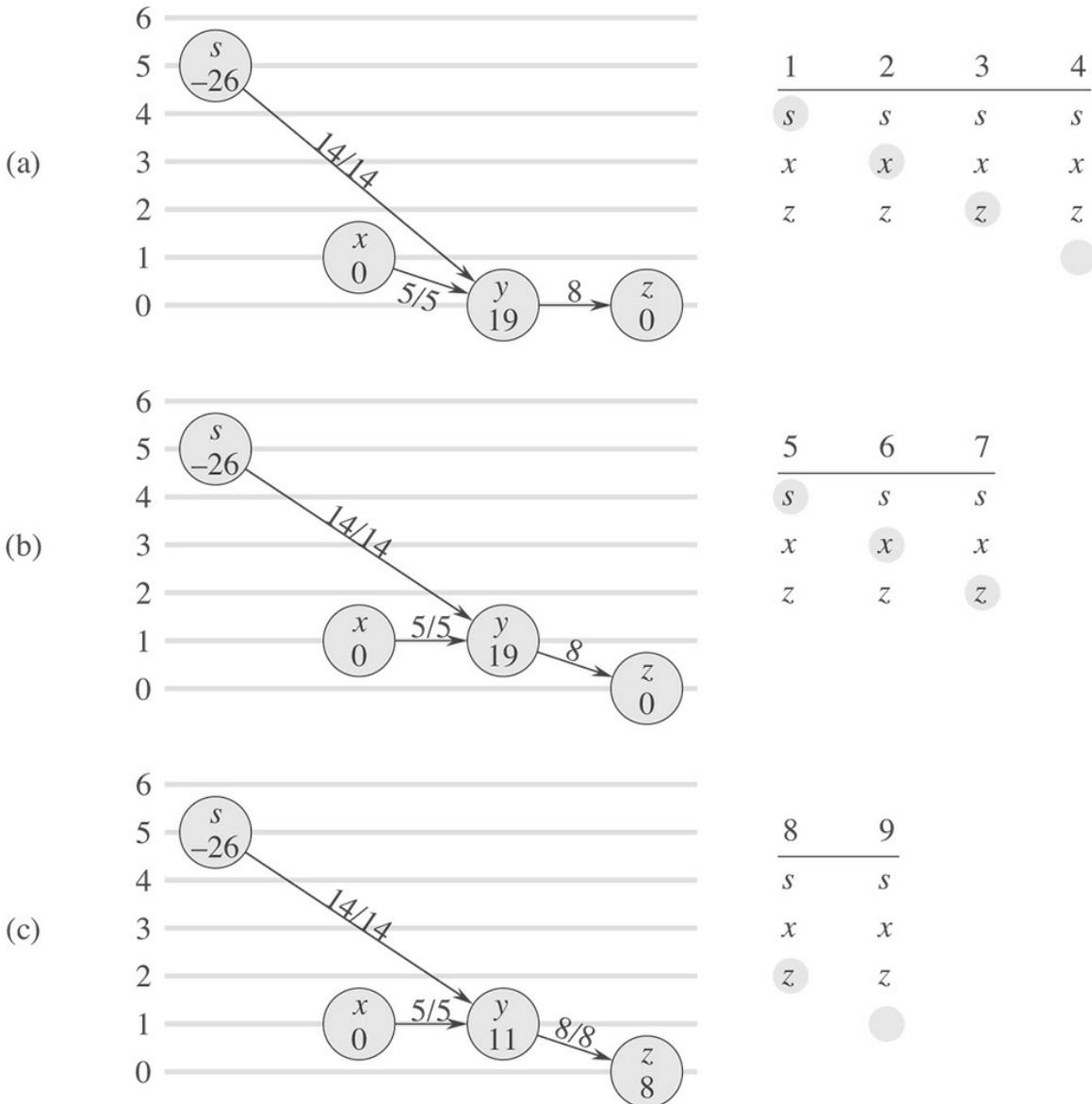
Para mostrar que nenhuma aresta admissível entra em u após uma operação relabel, suponha que exista um vértice v tal que (v, u) é admissível. Então, $v.h = u.h + 1$ após a operação relabel e, assim, $v.h > u.h + 1$ imediatamente antes de relabel. Porém, pelo Lema 26.12, não existe nenhuma aresta residual entre vértices cuja diferença entre as respectivas alturas seja mais de 1. Além disso, renomear um vértice não muda a rede residual. Portanto, (v, u) não está na rede residual e, consequentemente, não pode estar na rede admissível.

Listas de vizinhos

No algoritmo relabel-to-front, arestas são organizadas em "listas de vizinhos". Dada uma rede de fluxo $G = (V, E)$, a ***lista de vizinhos*** $u.N$ para um vértice $u \in V$ é uma lista simplesmente ligada dos vizinhos de u em G. Assim, o vértice v aparece na lista $u.N$ se $(u, v) \in E$ ou $(v, u) \in E$. A lista de vizinhos $u.N$ contém exatamente os vértices v para os quais pode existir uma aresta residual (u, v). O atributo *u.N.início* aponta para o primeiro vértice em $u.N$ e *v.próximo* aponta para o vértice que vem depois de v em uma lista de vizinhos; esse ponteiro é NIL se v é o último vértice na lista de vizinhos.

O algoritmo relabel-to-front percorre cada lista de vizinhos em ciclos e em uma ordem arbitrária fixa durante a execução do algoritmo. Para cada vértice u, o atributo *u.atual* aponta para o vértice que está sendo considerado em $u.N$ no momento em questão. Inicialmente, *u.atual* é definido como *u.N.início*.

Descarregando um vértice em transbordamento

Um vértice u é ***descarregado*** empurrando todo o seu excesso de fluxo por arestas admissíveis para vértices vizinhos; remarcando u conforme necessário para transformar as arestas que saem de u em arestas admissíveis. O pseudocódigo é dado a seguir.

```
DISCHARGE(u)
1  while u.e > 0
2      v = u.atual
3      if v == NIL
4          RELABEL(u)
5          u.atual = u.N.início
6      elseif c_f(u, v) > 0 e u.h = v.h + 1
7          PUSH(u, v)
8      else u.atual = v.próximo
```

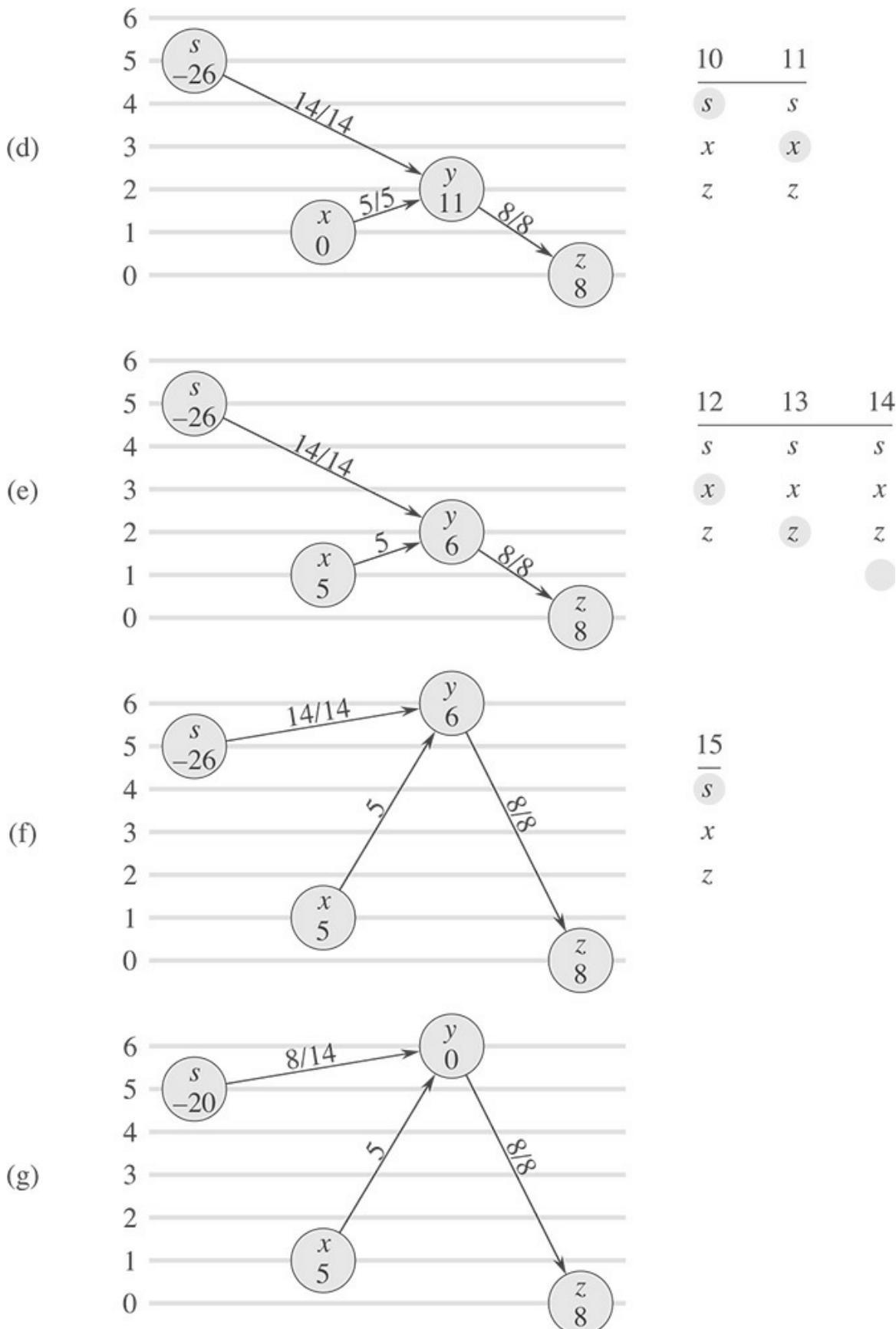



Figura 26.9 Descarregando um vértice y. São necessárias 15 iterações do laço **while** de DISCHARGE para empurrar todo o excesso de fluxo de y. Somente os vizinhos de y e as arestas que entram ou saem de y são mostrados. Em cada parte da figura, o número dentro de cada vértice é seu excesso no início da primeira iteração apresentada na parte, e cada vértice é mostrado na altura que ocupa nessa parte. A

lista de vizinhos $y.N$ no início de cada iteração aparece à direita, com o número da iteração na parte superior. O vizinho sombreado é $y.atual$. (**a**) Inicialmente, existem 19 unidades de excesso para empurrar de y e $y.atual = s$. As iterações 1, 2 e 3 simplesmente avançam $y.atual$, já que não existe nenhuma aresta admissível saindo de y. Na iteração 4, $y.atual$ = NIL (mostrado pelo sombreado abaixo da lista de vizinhos) e, assim, y é remarcado e $y.atual$ é redefinido como o início da lista de vizinhos. (**b**) Após a remarcação, o vértice y tem altura 1. Nas iterações 5 e 6, descobrimos que as arestas (y, s) e (y, x) são inadmissíveis, mas a iteração 7 empurra oito unidades de excesso de fluxo de y para z. Por causa do empurrão, $y.atual$ não avança nessa iteração. (**c**) Visto que o empurrão na iteração 7 saturou a aresta (y, z), descobrimos que ela é inadmissível na iteração 8. Na iteração 9, $y.atual$ = NIL e, assim, o vértice y é novamente remarcado e $y.atual$ é redefinido.

Figura 26.9 continuação (**d**) Na iteração 10, (y, s) é inadmissível, mas a iteração 11 empurra cinco unidades de excesso de fluxo de y para x. (**e**) Como *y.atual* não avançou na iteração 11, a iteração 12 descobre que (y, x) é inadmissível. A iteração 13 descobre que (y, z) é inadmissível, e a iteração 14 remarca o vértice y e redefine *y.atual*. (**f**) A iteração 15 empurra seis unidades de excesso de fluxo de y para s. (**g**) Agora o vértice y não tem nenhum excesso de fluxo e DISCHARGE termina. Neste exemplo, DISCHARGE começa e também termina com o ponteiro atual no início da lista de vizinhos mas, em geral, isso não é necessário.

A Figura 26.9 percorre explicitamente várias iterações do laço **while** das linhas 1–8, que é executado enquanto o vértice u tiver excesso positivo. Cada iteração executa exatamente uma de três ações, dependendo do vértice atual v na lista de vizinhos $u.N$.

1. Se v é NIL, passamos do fim de $u.N$. A linha 4 renomeia o vértice u e então a linha 5 redefine o vizinho atual de u como o primeiro em $u.N$. (O Lema 26.29 apresentado mais adiante afirma que a operação remarcação se aplica nessa situação.)
2. Se v é não NIL e (u, v) é uma aresta admissível (determinada pelo teste na linha 6), então a linha 7 empurra um pouco do excesso (ou possivelmente todo o excesso) de u para o vértice v.
3. Se v é não NIL, mas (u, v) é inadmissível, então a linha 8 ***avança*** $u.atual$ mais uma posição na lista de vizinhos $u.N$.

Observe que se DISCHARGE é chamada em um vértice u que está transbordando, então a última ação executada por DISCHARGE deve ser um empurrão a partir de u. Por quê? O procedimento termina somente quando $u.e$ se torna zero, e nem a operação remarcação nem o avanço do ponteiro $u.atual$ afeta o valor de $u.e$.

Temos de ter certeza de que, quando PUSH ou RELABEL é chamado por DISCHARGE a operação se aplica. O próximo lema prova esse fato.

Lema 26.29

Se DISCHARGE chama PUSH(u, v) na linha 7, então uma operação PUSH se aplica a (u, v). Se DISCHARGE chama RELABEL(u) na linha 4, então uma operação de remarcação se aplica a u.

Prova Os testes das linhas 1 e 6 asseguram que uma operação empurrão ocorre somente se a operação se aplicar, o que prova a primeira declaração no lema.

Para provar a segunda declaração, de acordo com o teste na linha 1 e com o Lema 26.28, basta mostrar que todas as arestas que saem de u são inadmissíveis. Se uma chamada a DISCHARGE(u) começa com o ponteiro $u.atual$ no início da lista de vizinhos de u e termina com ele fora do final da lista, então todas as arestas que saem de u são inadmissíveis e uma operação remarcação se aplica. Todavia, é possível que durante uma chamada a DISCHARGE(u) o ponteiro $u.atual$ percorra somente parte da lista antes de o procedimento retornar. Então podem ocorrer chamadas a DISCHARGE em outros vértices, mas $u.atual$ continuará se movendo pela lista durante a próxima chamada a DISCHARGE(u). Agora consideramos o que acontece durante uma passagem completa pela lista, que começa no início de $u.N$ e termina com $u.atual$ = NIL. Tão logo $u.atual$ chegue ao final da lista, o procedimento renomeia u *e* começa uma nova passagem. Para o ponteiro $u.atual$ ultrapassar um vértice $v \in u.N$ durante uma passagem, a aresta (u,v) deve ser considerada inadmissível pelo teste na linha 6. Assim, quando a passagem é concluída, toda aresta que sai de u já foi determinada como inadmissível em algum momento durante a passagem. A observação fundamental é que, no final da passagem, toda aresta que sai de u ainda é inadmissível. Por quê? Pelo Lema 26.27, empurrões não podem criar nenhuma aresta admissível, independentemente do vértice do qual o fluxo é empurrado. Assim, qualquer aresta admissível deve ser criada por uma operação remarcação. Porém, o vértice u não é remarcado durante a passagem e, pelo Lema 26.28, qualquer outro vértice v que é renomeado durante a passagem (resultante de uma chamada de DISCHARGE(v)) não tem nenhuma aresta de entrada admissível após a renomeação. Assim, ao final da passagem, todas as arestas que saem de u continuam inadmissíveis, o que conclui a prova.

O algoritmo relabel-to-front

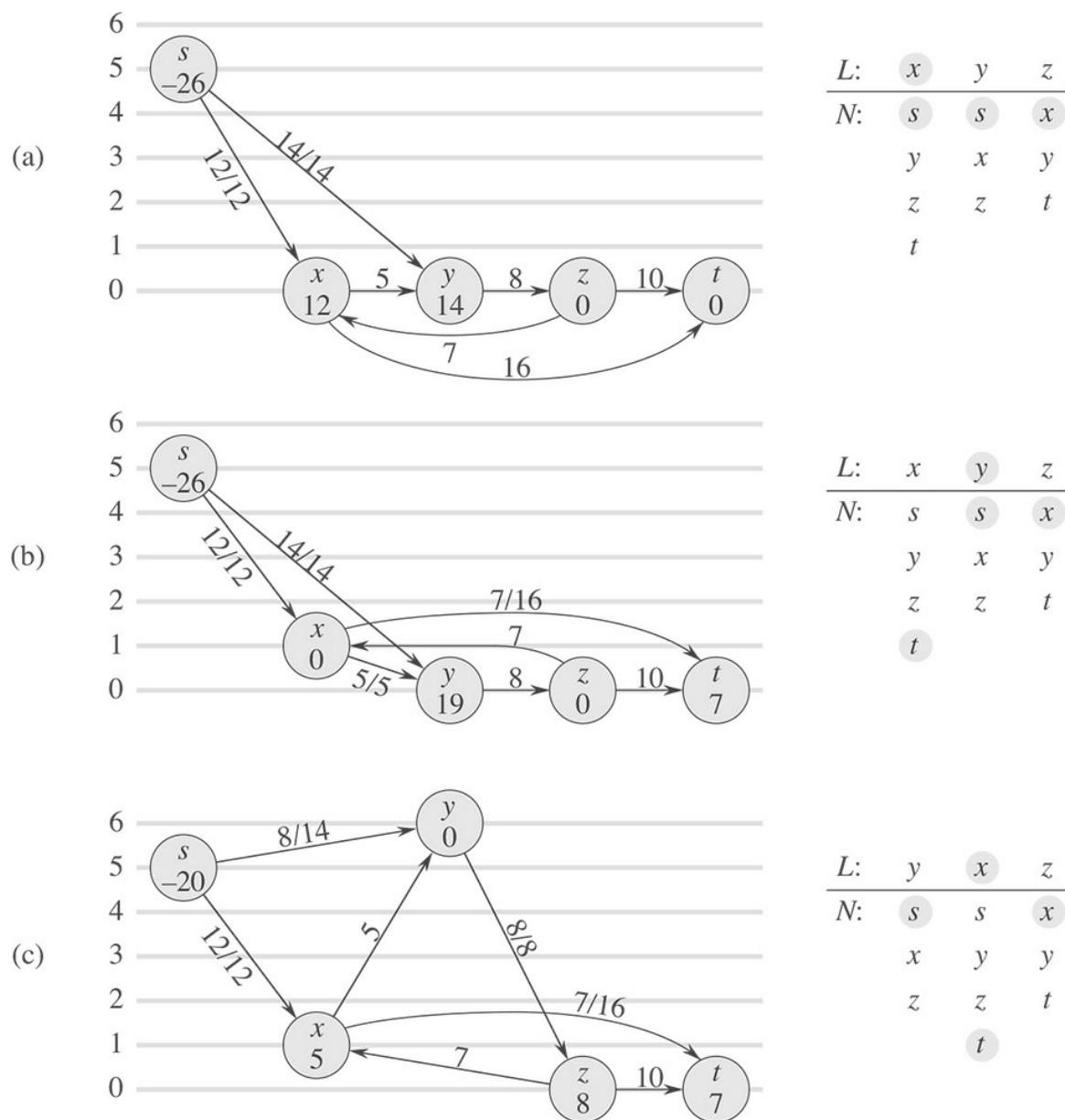
No algoritmo relabel-to-front, mantemos uma lista ligada L que consiste em todos os vértices em $V - \{s, t\}$. Uma propriedade fundamental é que os vértices em L são ordenados topologicamente de acordo com a rede admissível, como veremos no invariante de laço a seguir. (Lembre-se de que vimos no Lema 26.26 que a rede admissível é um gad.)

O pseudocódigo para o algoritmo relabel-to-front supõe que as listas de vizinhos $u.N$ já foram criadas para cada vértice u. Supõe também que $u.próximo$ aponta para o vértice que vem depois de u na lista L e que, como sempre, $u.próximo$ = NIL se u é o último vértice na lista.

```
RELABEL-TO-FRONT(G, s, t)

1     INITIALIZE-PREFLOW(G, s)
2     L = G.V − {s, t}, em qualquer ordem
3     for cada vértice u ∈ G.V − {s, t}
4         u.atual = u.N.início
5     u = L.início
6     while u ≠ NIL
7         altura-antiga = u.h
8         DISCHARGE(u)
9         if u.h > altura-antiga
10                mover u para a frente da lista L
11        u = u.próximo
```

O algoritmo relabel-to-front funciona da maneira descrita a seguir. A linha 1 inicializa o pré-fluxo e as alturas para os mesmos valores que no algoritmo push-relabel genérico. A linha 2 inicializa a lista L para conter todos os vértices que potencialmente estariam transbordando, em qualquer ordem. As linhas 3–4 inicializam o ponteiro *atual* de cada vértice u como o primeiro vértice na lista de vizinhos de u.

Como mostra a Figura 26.10, o laço **while** das linhas 6–11 percorre a lista L, descarregando os vértices. A linha 5 o obriga a começar no primeiro vértice na lista. A cada passagem pelo laço, a linha 8 descarrega um vértice u. Se u foi remarcado pelo procedimento DISCHARGE, a linha 10 o desloca para a frente da lista L. Podemos determinar se u foi remarcado comparando sua altura antes da operação de descarga, que foi gravada na variável *altura-antiga* na linha 7, com sua altura depois da operação de descarga, na linha 9. A linha 11 obriga a próxima iteração do laço **while** a usar o vértice que vem depois de u na lista L. Se a linha 10 deslocou u para a frente da lista, o vértice usado na próxima iteração é aquele que vem depois de u em sua nova posição na lista.

Para mostrar que RELABEL-TO-FRONT calcula um fluxo máximo, mostraremos que ele é uma implementação do algoritmo push-relabel genérico. Primeiro, observe que ele executa operações remarcação e empurrão só quando elas são aplicáveis, já que o Lema 26.29 garante que DISCHARGE só executará essas operações quando eles se aplicarem. Resta mostrar que, quando RELABEL-TO-FRONT termina, nenhuma operação básica se aplica. O restante do argumento de correção se baseia no seguinte invariante de laço:

Em cada teste na linha 6 de RELABEL-TO-FRONT, a lista L é uma ordenação topológica dos vértices na rede admissível $G_{f,h} = (V, E_{f,h})$, e nenhum vértice antes de u na lista tem excesso de fluxo.

Inicialização: Imediatamente após a execução de INITIALIZE-PREFLOW, $s.h = |V|$ e $v.h = 0$ para todo $v \in V - \{s\}$. Visto que $|V| \geq 2$ (porque V contém no mínimo s e t), nenhuma aresta pode ser admissível. Assim, $E_{f,h} = \emptyset$, e qualquer ordenação de $V - \{s, t\}$ é uma ordenação topológica de $G_{f,h}$. Como inicialmente u está no começo da lista L, não existe nenhum vértice antes dele, e portanto não há nenhum antes dele com excesso de fluxo.

Manutenção: Para ver que cada iteração do laço **while** mantém a ordenação topológica, começamos observando que a rede admissível é mudada somente por operações empurrão e remarcação. Pelo Lema 26.27, as operações empurrão não transformam arestas em admissíveis. Assim, somente operações remarcação podem criar arestas admissíveis. Todavia, depois que um vértice u é remarcado, o Lema 26.28 afirma que não existe

nenhuma aresta admissível entrando em u, mas pode haver arestas admissíveis saindo de u. Assim, deslocando u para a frente de L, o algoritmo garante que quaisquer arestas admissíveis que saem de u satisfazem a ordenação topológica. Para ver que nenhum vértice que precede u em L tem excesso de fluxo, denotamos por u o vértice que será u' na próxima iteração. Os vértices que precederão u' na próxima iteração incluem o u atual (devido à linha 11) e nenhum outro vértice (se u for remarcado) ou os mesmos vértices de antes (se u não for remarcado). Quando u é descarregado, não tem nenhum excesso de fluxo depois. Assim, se u for remarcado durante a descarga, nenhum vértice que precede u' terá excesso de fluxo. Se u não for remarcado durante a descarga, nenhum vértice antes dele na lista adquiriu excesso de fluxo durante essa descarga porque L permaneceu ordenada topologicamente o tempo todo durante a descarga (conforme acabamos de destacar, arestas admissíveis são criadas somente por remarcação, e não por empurrão) e, portanto, cada operação empurrão obriga o excesso de fluxo a se deslocar somente para vértices que estão mais abaixo na lista (ou para s ou t). Novamente, nenhum vértice que precede u' tem excesso de fluxo.

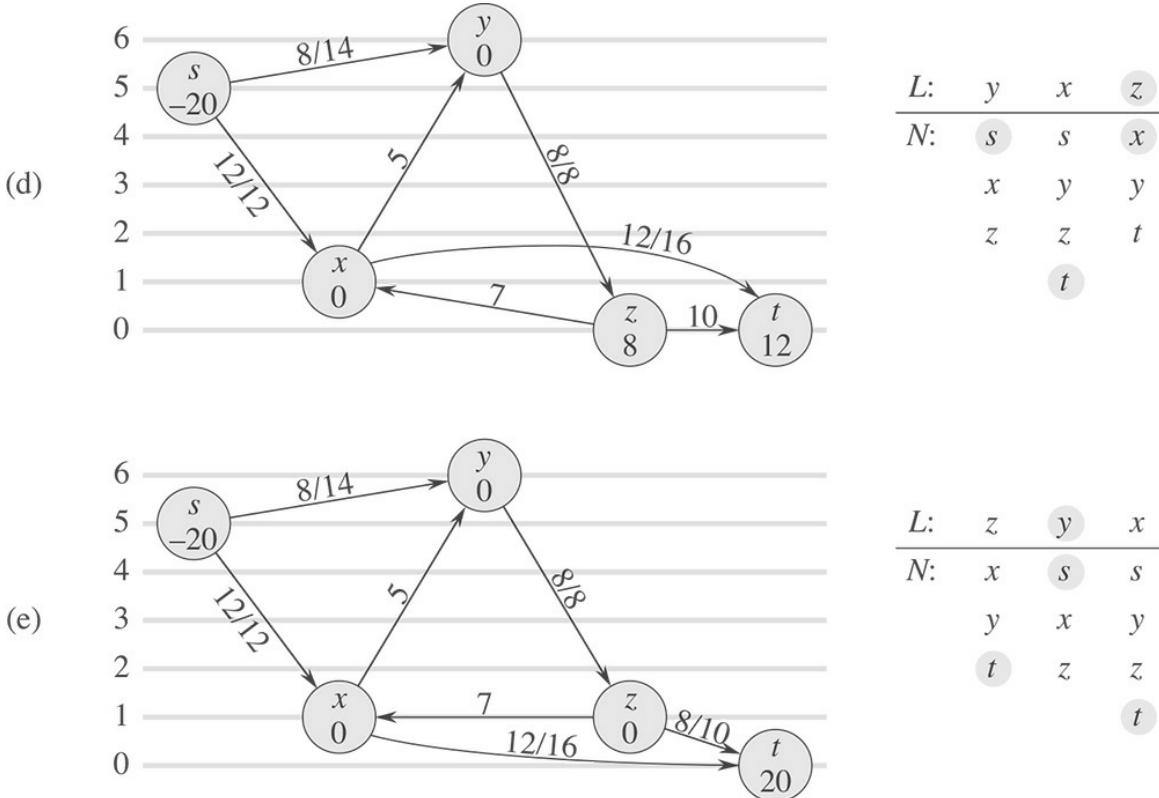



Figura 26.10 A ação de RELABEL-TO-FRONT. (**a**) Uma rede de fluxo imediatamente antes da primeira iteração do laço **while**. No início, 26 unidades de fluxo saem da fonte s. À direita é mostrada a lista inicial $L = \langle x, y, z \rangle$, onde inicialmente $u = x$. Sob cada vértice na lista L está sua lista de vizinhos, com o vizinho atual sombreado. O vértice x é descarregado. Ele é renomeado para altura 1, cinco unidades de excesso de fluxo são empurradas para y, e as sete unidades de excesso restantes são empurradas para o sorvedouro t. Como x foi

remarcado, ele se desloca para o início de L, o que nesse caso não muda a estrutura de L. (**b**) Depois de x, y *é* o próximo vértice em L que é descarregado. A Figura 26.9 mostra a ação detalhada de descarregar y nessa situação. Como y foi remarcado, ele passa para o início de L. (**c**) Agora o vértice x vem depois de y em L, e portanto ele é novamente descarregado, empurrando todas as cinco unidades de excesso de fluxo para t. Como o vértice x não é remarcado nessa operação de descarga, ele permanece em seu lugar na lista L.

Término: Quando o laço termina, u acabou de passar do final de L e, portanto, o invariante de laço garante que o excesso de todo vértice é 0. Assim, nenhuma operação básica se aplica.

Análise

Mostraremos agora que Relabel-To-Front é executado no tempo $O(V_3)$ em qualquer rede de fluxo $G = (V, E)$**.** Visto que o algoritmo é uma implementação do algoritmo push-relabel genérico, aproveitaremos o Corolário 26.21, que dá um limite $O(V)$ para o número de operações remarcação executadas por vértice e um limite $O(V_2)$ para o número total global de operações remarcação. Além disso, o Exercício 26.4–3 dá um limite $O(VE)$ para o tempo total gasto na execução de operações remarcação, e o Lema 26.22 dá um limite $O(VE)$ para o número total de operações empurrões saturadores.

Figura 26.10 continuação (**d**) Visto que o vértice z vem depois do vértice x em L, ele é descarregado. O vértice é remarcado para a altura 1 e todas as oito unidades de excesso de fluxo são empurradas para t. Como z é remarcado, passa para a frente de L. (**e**) O vértice y agora vem depois do vértice z em L e, portanto, é descarregado. Porém, como y não tem nenhum excesso, Discharge retorna imediatamente e y permanece em seu lugar em L. O vértice x é então descarregado. Como também ele não tem nenhum excesso, DISCHARGE retorna mais uma vez e x permanece em seu lugar em L. Relabel-To-FronT chegou ao final da lista L e termina. Não existe nenhum vértice que está transbordando, e o pré-fluxo é um fluxo máximo.

Teorema 26.30

O tempo de execução de Relabel-To-Front em qualquer rede de fluxo $G = (V, E)$ é $O(V_3)$.

Prova Vamos considerar uma "fase" do algoritmo relabel-to-front como o tempo entre duas operações remarcação consecutivas. Há $O(V_2)$ fases, já que há $O(V_2)$ operações remarcação. Cada fase consiste em no máximo $|V|$ chamadas a DISCHARGE, o que podemos ver a seguir. Se DISCHARGE não executar uma operação remarcação, a próxima chamada a DISCHARGE está ainda mais embaixo na lista L, e o comprimento de L é menor que $|V|$. Se DISCHARGE executar uma operação remarcação, a próxima chamada a DISCHARGE pertence a uma fase diferente. Visto que cada fase contém no máximo $|V|$ chamadas a DISCHARGE E há $O(V_2)$ fases, o número de vezes que DISCHARGE É chamada na linha 8 de RELABEL-TO-FRONT É $O(V_3)$. Assim, o trabalho total realizado pelo laço **while** em RELABEL-TO-FRONT, excluindo o trabalho executado dentro de DISCHARGE, é no máximo $O(V_3)$.

Agora, devemos limitar o trabalho realizado dentro de DISCHARGE durante a execução do algoritmo. Cada iteração do laço **while** dentro de DISCHARGE executa uma de três ações. Analisaremos a quantidade total de trabalho envolvido na execução de cada uma dessas ações.

Começamos com operações remarcação (linhas 4–5). O Exercício 26.4–3 dá um limite de tempo $O(VE)$ para todas as $O(V_2)$ operações remarcação executadas.

Agora, suponha que a ação atualize o ponteiro *u.atual* na linha 8. Essa ação ocorre $O(\text{grau}(u))$ vezes cada vez que um vértice u é renomeado e, no geral, $O(V \cdot \text{grau}(u))$ vezes para o vértice. Portanto, para todos os vértices, a quantidade total de trabalho realizado para avançar ponteiros em listas de vizinhos é $O(VE)$ pelo lema do aperto de mão (Exercício B.4–1).

O terceiro tipo de ação executada por DISCHARGE é uma operação empurrão (linha 7). Já sabemos que o número total de operações empurrão saturador é $O(VE)$. Observe que, se um empurrão não saturador é executado, DISCHARGE retorna imediatamente, já que o empurrão reduz o excesso a 0. Assim, pode haver no máximo um empurrão não saturador por chamada a DISCHARGE. Como observamos, DISCHARGE é chamada $O(V_3)$ vezes, e assim o tempo total gasto na execução de empurrões não saturadores é $O(V_3)$.

Portanto, o tempo de execução de RELABEL-TO-FRONT é $O(V_3 + VE)$, que é $O(V_3)$.

Exercícios

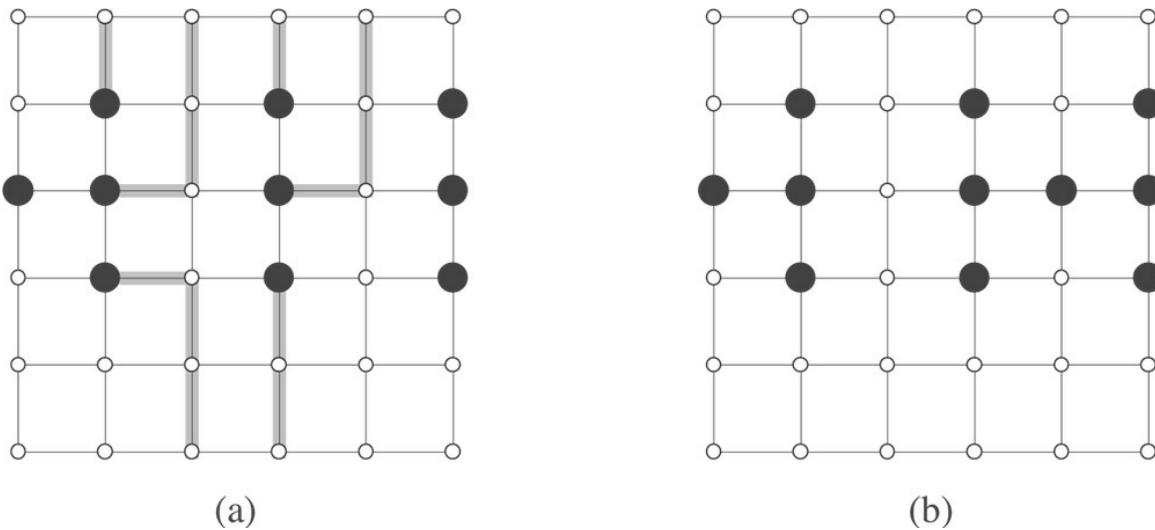
26.5-1 Ilustre a execução de RELABEL-TO-FRONT à maneira da Figura 26.10 para a rede de fluxo da Figura 26.1(a). Considere que a ordenação inicial de vértices em L seja $\langle v_1, v_2, v_3, v_4\rangle$ e que as listas de vizinhos sejam

$$v_1.N = \langle s, v_2, v_3\rangle,$$
$$v_2.N = \langle s, v_1, v_3, v_4\rangle,$$
$$v_3.N = \langle v_1, v_2, v_4, t\rangle,$$
$$v_4.N = \langle v_2, v_3, t\rangle.$$

26.5-2 ★ Gostaríamos de implementar um algoritmo push-relabel no qual mantivéssemos uma fila do tipo primeiro a entrar, primeiro a sair de vértices que estão transbordando. O algoritmo descarrega repetidamen te o vértice que está no início da fila e quaisquer vértices que não estavam transbordando antes da descarga mas que estão transbordando depois são colocados no final da fila. Depois de descarregado, o vértice que está no início da fila é removido. Quando a fila está vazia, o algoritmo termina. Mostre como implementar esse algoritmo para calcular um fluxo máximo no tempo $O(V_3)$.

26.5-3 Mostre que o algoritmo genérico ainda funciona se RELABEL atualiza $u.h$ simplesmente calculando $u.h = u.h + 1$. Como essa mudança afetaria a análise de RELABEL-TO-FRONT?

26.5-4 ★Mostre que, se sempre descarregarmos o vértice mais alto que está transbordando, podemos conseguir que o método push-relabel seja executado no tempo $O(V_3)$.

26.5-5 Suponha que em algum ponto na execução de um algoritmo push-relabel, exista um inteiro $0 < k \leq |V| - 1$ para o qual nenhum vértice tem $v.h = k$. Mostre que todos os vértices com $v.h > k$ estão no lado da fonte de um corte mínimo. Se tal k existir, a ***heurística de lacuna*** atualiza todo vértice $v \in V - \{s\}$ para o qual $v.h > k$ para definir $v.h = \max(v.h, |V| + 1)$. Mostre que o atributo h resultante é uma função altura. (A heurística de lacuna é crucial para conseguir que as implementações do método push-relabel funcionem bem na prática.)

Problemas

26-1 ***O problema do escape***

Uma ***grade*** $n \times n$ é um grafo não dirigido que consiste em n linhas e n colunas de vértices, como mostra a Figura 26.11. Denotamos o vértice na i-ésima linha e j-ésima coluna por (i, j). Todos os vértices em uma grade têm exatamente quatro vizinhos, exceto os vértices do contorno, que são os pontos (i, j) para os quais $i = 1$, $i = n$, $j = 1$ ou $j = n$.

Dados $m \leq n_2$ pontos de partida $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$ na grade, o ***problema do escape*** consiste em determinar se existem ou não m caminhos disjuntos de vértices dos pontos de partida a quaisquer m pontos diferentes no contorno. Por exemplo, a grade na Figura 26.11(a) tem um escape, mas a grade na Figura 26.11(b) não tem.

Figura 26.11 Grades para o problema do escape. Os pontos de partida são pretos, e os outros vértices da grade são brancos. (**a**) Uma grade com escape, mostrado por caminhos sombreados. (**b**) Uma grade sem escape.

a Considere uma rede de fluxo na qual os vértices, bem como as arestas, têm capacidades. Isto é, o fluxo positivo total que entra em qualquer vértice dado está sujeito a uma restrição de capacidade. Mostre que determinar o fluxo máximo em uma rede com capacidades de arestas e vértices pode ser reduzido a um problema comum de fluxo máximo em uma rede de fluxo de tamanho comparável.

b Descreva um algoritmo eficiente para resolver o problema do escape e analise seu tempo de execução.

26-2 ***Cobertura de caminhos mínima***

Uma ***cobertura de caminhos*** de um grafo dirigido $G = (V, E)$ é um conjunto P de caminhos disjuntos nos vértices tal que todo vértice em V está incluído em exatamente um caminho em P. Os caminhos podem começar e terminar em qualquer lugar e ter qualquer comprimento, inclusive 0.

Uma ***cobertura de caminho mínima*** de G é uma cobertura de caminho que contém o menor número possível de caminhos.

a Dê um algoritmo eficiente para encontrar uma cobertura de caminho mínima de um grafo acíclico dirigido $G = (V, E)$. (*Sugestão*: Considerando que $V = \{1, 2, ..., n\}$, construa o grafo $G' = (V', E')$, onde

$$V' = \{x_0, x_1, ..., x_n\} \cup \{y_0, y_1, ..., y_n\} ,$$
$$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\} ,$$

e execute um algoritmo de fluxo máximo.)

b Seu algoritmo funciona para grafos dirigidos que contêm ciclos? Explique.

26-3 Consultoria de algoritmos

O professor Gore quer abrir uma empresa de consultoria de algoritmos. Ele identificou n subáreas importantes de algoritmos (que correspondem aproximadamente às diferentes partes deste livro) e as representa pelo conjunto $A = \langle A_1, A_2, ..., A_n \rangle$. Para cada subárea A_k, ele pode contratar um profissional especializado nessa área por c_k dólares. A empresa de consultoria alinhou um conjunto $J = \{J_1, J_2, ..., J_m\}$ de serviços potenciais.

Para executar o serviço J_i, a empresa precisaria ter contratado profissionais em um subconjunto $R_i \subseteq A$ de subáreas. Cada profissional pode trabalhar em vários serviços simultaneamente. Se a empresa optar por aceitar o trabalho J_i, deve ter contratado profissionais em todas as subáreas em R_i, e conseguirá uma receita de p_i dólares.

O trabalho do professor Gore é determinar para quais subáreas contratar profissionais e quais serviços aceitar, de modo a maximizar a receita líquida, que é a receita total auferida dos trabalhos aceitos menos o custo de empregar os profissionais especializados. Considere a seguinte rede de fluxo G. Ela contém um vértice de fonte s, vértices $A_1, A_2, ...,A_n$, vértices $J_1, J_2, ...,J_n$ e um vértice de sorvedouro t. Para $k = 1, 2 ..., n$, a rede de fluxo contém uma aresta (s, A_k) com capacidade $c(s, A_k) = c_k$ e, para $i = 1, 2 ..., n$, a rede de fluxo contém uma aresta (J_i, t) com capacidade $c(J_i, t) = p_i$. Para $k = 1, 2, ..., n$ e $i = 1, 2, ..., m$, se $A_k \in R_i$, então G contém uma aresta (A_k, J_i) com capacidade $c(A_k, J_i) = \infty$.

a Mostre que, se $J_i \in T$ para um corte de capacidade finita (S, T) de G, então $A_k \in T$ para cada $A_k \in R_i$.

b Mostre como determinar a receita líquida máxima pela capacidade de um corte mínimo de G e valores p_i dados.

c Dê um algoritmo eficiente para determinar quais serviços aceitar e quais profissionais especializados contratar. Analise o tempo de execução do seu algoritmo em termos de m, n e r $\sum_{i=1}^{m} |R_i|$

26-4 Atualização do fluxo máximo

Seja $G = (V, E)$ uma rede de fluxo com fonte s, sorvedouro t e capacidades inteiras.

Suponha que temos um fluxo máximo em G.

a Suponha que aumentamos de 1 a capacidade de uma única aresta $(u, v) \in E$. Dê um algoritmo de tempo $O(V + E)$ para atualizar o fluxo máximo.

b Suponha que reduzimos de 1 a capacidade de uma única aresta $(u, v) \in E$. Dê um algoritmo de tempo $O(V + E)$ para atualizar o fluxo máximo.

26-5 Fluxo máximo por mudança de escala

Seja $G = (V, E)$ uma rede de fluxo com fonte s, sorvedouro t e uma capacidade inteira $c(u, v)$ em cada aresta $(u, v) \in E$. Seja $C = \max_{(u, v) \in E} c(u, v)$.

a Demonstre que um corte mínimo de G tem no máximo capacidade $C|E|$.

b Para um dado número K, mostre como determinar um caminho aumentador de capacidade no mínimo K no tempo $O(E)$, se tal caminho existir.

Podemos usar a seguinte modificação de FORD-FULKERSON-METHOD para calcular um fluxo máximo em G:

```
MAX-FLOW-BY-SCALING(G, s, t)
1 C = max_{(u,v)∈E} c(u, v)
2 inicializar fluxo f como 0
3 K = 2^⌊lg C⌋
4 while K ≥ 1
5    while existir um caminho aumentador p de capacidade no mínimo K
6        aumentar o fluxo f ao longo de p
7    K = K/2
8 return f
```

c Demonstre que MAX-FLOW-BY-SCALING retorna um fluxo máximo.

d Mostre que a capacidade de um corte mínimo da rede residual G_f é no máximo $2K|E|$ cada vez que a linha 4 é executada.

e Demonstre que o laço **while** interno das linhas 5-6 é executado $O(E)$ vezes para cada valor de K.

f Conclua que MAX-FLOW-BY-SCALING pode ser implementado de modo que seja executado no tempo $O(E^2 \lg C)$.

26-6 Algoritmo Hopcroft-Karp para emparelhamento de grafo bipartido

Neste problema, descrevemos um algoritmo mais rápido, criado por Hopcroft e Karp, para encontrar um emparelhamento máximo em um grafo bipartido. O algoritmo funciona no tempo $O(\sqrt{E})$. Dado um grafo não dirigido bipartido $G = (V, E)$, onde $V = L \cup R$ e todas as arestas têm exatamente uma extremidade em L, seja M um emparelhamento em G. Dizemos que um caminho simples P em G é um ***caminho aumentador*** para a M se ele começa em um vértice não emparelhado em L, termina em um vértice não emparelhado em R e suas arestas pertencem alternadamente a M e a $E - M$. (Essa definição de um caminho aumentador está relacionada, embora seja diferente, com um caminho aumentador em uma rede de fluxo.) Neste problema, tratamos um caminho como uma sequência de arestas, em vez de uma sequência de vértices. Um caminho

mínimo aumentador referente a um emparelhamento M é um caminho aumentador que tem um número mínimo de arestas.

Dados dois conjuntos A e B, a ***diferença simétrica*** $A \oplus B$ é definida como $(A - B) \cup (B - A)$, isto é, os elementos que estão em exatamente um dos dois conjuntos.

a Mostre que, se M é um emparelhamento e P é um caminho aumentador para M, então a diferença simétrica $M \oplus P$ é um emparelhamento e $|M \oplus P| = |M| + 1$. Mostre que, se $P_1, P_2, ..., P_k$ são caminhos aumentadores em relação a M disjuntos nos vértices, então a diferença simétrica $M \oplus (P_1 \cup P_2 \cup ... \cup P_k)$ é um emparelhamento com cardinalidade $|M| + k$.

A estrutura geral de nosso algoritmo é a seguinte:

```
HOPCROFT-KARP(G)

1 M = ∅
2 repeat
3     seja 𝒫 = {P_1, P_2, ..., P_k} um conjunto máximo de
          caminhos mínimos aumentadores disjuntos nos vértices para M
4     M = M ⊕ (P_1 ∪ P_2 ∪ ... ∪ P_k)
5 until 𝒫 == ∅
6 return M
```

O restante desse problema pede que você analise o número de iterações no algoritmo (isto é, o número de iterações no laço **repeat**) e descreva uma implementação da linha 3.

b Dados dois emparelhamentos M e M_* em G, mostre que todo vértice no grafo $G' = (V, M \oplus M_*)$ tem grau no máximo 2. Conclua que G' é uma união disjunta de caminhos ou ciclos simples. Demonstre que arestas em cada um desses caminhos ou ciclos simples pertencem alternadamente a M ou M_*. Prove que, se $|M| \leq |M_*|$, então $M \oplus M_*$ contém no mínimo $|M_*| - |M|$ caminhos aumentadores disjuntos nos vértices para M.

Seja l o comprimento de um caminho mínimo aumentador para um emparelhamento M e seja $P_1, P_2, ..., P_k$ um conjunto máximo de caminhos aumentadores para M, disjuntos nos vértices, de comprimento l. Seja $M' = M \oplus (P_1 \cup ... \cup P_k)$ e suponha que P seja um caminho mínimo aumentador para M'.

c Mostre que, se P é disjuntos nos vértices de $P_1, P_2, ..., P_k$, então P tem mais de l arestas.

d Agora, suponha que P não é disjuntos nos vértices de $P_1, P_2, ..., P_k$. Seja A o conjunto de arestas $(M \oplus M') \oplus P$. Mostre que $A = (P_1 \cup P_2 \cup ... \cup P_k) \oplus P$ e que $|A| \geq (k + 1)l$. Conclua que P tem mais de l arestas.

e Prove que, se um caminho aumentador mínimo para M tem l arestas, o tamanho do emparelhamento máximo é no máximo $|M| + |V|/l$.

f Mostre que o número de iterações do laço **repeat** no algoritmo é no máximo $\sqrt{2}V$ (*Sugestão:* De quanto M pode crescer após a iteração número $\sqrt{V}$)

g Dê um algoritmo que funcione no tempo $O(E)$ para encontrar um conjunto máximo de caminhos mínimos aumentadores disjuntos nos vértices $P_1, P_2, ..., P_k$ para um dado emparelhamento M. Conclua que o tempo de execução total de HOPCROFT–KARP é $O(\sqrt{V}E)$.

NOTAS DO CAPÍTULO

Ahuja, Magnanti e Orlin [7], Even [103], Lawler [224], Papadimitriou e Steiglitz [271] e ainda Tarjan [330] são boas referências para redes de fluxo e algoritmos relacionados. Goldberg, Tardos e Tarjan [139] também dão um ótimo levantamento de algoritmos para problemas de redes de fluxo, e Schrijver [304] escreveu uma resenha interessante de desenvolvimentos históricos na área de redes de fluxo.

O método Ford–Fulkerson se deve a Ford e a Fulkerson [109], que deram origem ao estudo formal de muitos dos problemas na área de redes de fluxo, incluindo os problemas de fluxo máximo e emparelhamento em grafo bipartido. Muitas implementações iniciais do método de Ford–Fulkerson encontravam caminhos aumentadores usando busca em largura; Edmonds e Karp [102] e, independentemente, Dinic [89], provaram que essa estratégia produz um algoritmo de tempo polinomial. Uma ideia relacionada, a de utilizar "fluxos bloqueadores", também foi desenvolvida primeiro por Dinic [89]. Karzanov [202] foi o primeiro a desenvolver a ideia de pré–fluxos. O método push-relabel se deve a Goldberg [136] e a Goldberg e Tarjan [140]. Goldberg e Tarjan apresentaram um algoritmo de tempo $O(V_3)$ que usa uma fila para manter o conjunto de vértices que estão transbordando, bem como um algoritmo que usa árvores dinâmicas para conseguir um tempo de execução de $O(V E \lg (V_2/E + 2))$. Vários outros pesquisadores desenvolveram algoritmos push–relabel de fluxo máximo. Ahuja e Orlin [9] e Ahuja, Orlin e Tarjan [10] deram algoritmos que usavam escalonamento. Cheriyan e Maheshwari [63] propuseram empurrar o fluxo a partir do vértice que está transbordando que tenha a altura máxima. Cheriyan e Hagerup [61] sugeriram permutar aleatoriamente a lista de vizinhos, e vários pesquisadores [14, 204, 276] desenvolveram desaleatorizações inteligentes dessa ideia, o que levou a uma sequência de algoritmos mais rápidos. O algoritmo de King, Rao e Tarjan [204] é o mais rápido desses algoritmos e funciona no tempo $O(V E \log_{E/(V \lg V)} V)$.

O algoritmo assintoticamente mais rápido criado até hoje para o problema de fluxo máximo, criado por Goldberg e Rao [138], funciona no tempo $O(\min(V_{2/3}, E_{1/2})\, E \lg(V^2/E + 2) \lg C)$, onde $C = \max_{(u, v) \in E} c(u, v)$. Esse algoritmo não usa o método push–relabel; em vez disso, é baseado na localização de fluxos de bloqueio. Todos os algoritmos de fluxo máximo anteriores, incluindo os que examinamos neste capítulo, usam alguma noção de distância (os algoritmos push–relabel utilizam a noção análoga de altura), sendo que um comprimento 1 é atribuído implicitamente a cada aresta. Esse novo algoritmo adota uma abordagem diferente e atribui comprimento 0 a arestas de alta capacidade e comprimento 1 a arestas de baixa capacidade. Informalmente, no que tange a esses comprimentos, caminhos mínimos da fonte ao sorvedouro tendem a ter capacidade alta, o que significa que é preciso executar um número menor de iterações.

Na prática, atualmente os algoritmos push-relabel dominam a área dos algoritmos de caminho aumentador ou dos algoritmos baseados em programação linear para o problema de fluxo máximo. Um estudo de Cherkassky e Goldberg [63] reforça a importância de usar duas heurísticas na implementação de um algoritmo push-relabel. A primeira heurística é executar periodicamente uma busca em largura da rede residual para obter valores mais precisos para a altura. A segunda heurística é a heurística da lacuna, descrita no Exercício 26.5–5. Cherkassky e Goldberg concluem que a melhor escolha de variantes do algoritmo push–relabel é a que opta por descarregar o vértice que está transbordando que tem a altura máxima.

O melhor algoritmo criado até hoje para o emparelhamento máximo em grafo bipartido, descoberto por Hopcroft e Karp [176], é executado no tempo $O(\sqrt{V}E)$ e descrito no Problema 26–6. O livro de Lovász e Plummer [239] é uma excelente referência para problemas de emparelhamento.

---

[1] Lembre-se de que, na Seção 22.1, representamos um atributo f para a aresta (u,v) com o mesmo estilo de notação — $(u,v) \cdot f$ — que usamos para um atributo de qualquer outro objeto.

[2] O método de Ford-Fulkerson pode não terminar no término somente se as capacidades de arestas forem números irracionais.

[3] Na literatura, uma função altura, normalmente é denominada "função distância", e a altura de um vértice é denominada "rótulo de distância". Usamos o termo "altura" porque é mais sugestivo da intuição que fundamenta o algoritmo. Conservamos o uso do termo "remarcar" (*relabel*) para referenciar a operação que aumenta a altura de um vértice. A altura de um vértice está relacionada com sua distância em relação ao sorvedouro t, como seria determinado por uma busca em largura do transposto G_T.

Parte

TÓPICOS SELECIONADOS

INTRODUÇÃO

Esta parte contém uma seleção de tópicos sobre algoritmos que amplia e complementa material já apresentado neste livro. Alguns capítulos apresentam novos modelos de computação, como circuitos ou computadores paralelos. Outros abrangem domínios especializados, como geometria computacional ou teoria dos números. Os dois últimos capítulos discutem algumas das limitações conhecidas para o projeto de algoritmos eficientes e técnicas para enfrentar essas limitações.

O Capítulo 27 apresenta um modelo algorítmico para computação paralela baseado em multithread dinâmico. O capítulo apresenta os aspectos básicos do modelo e mostra como quantificar paralelismo em termos de medidas de trabalho e duração. Então investiga vários algoritmos multithread interessantes, incluindo algoritmos para multiplicação de matrizes e ordenação por intercalação.

O Capítulo 28 estuda algoritmos eficientes para operações com matrizes. Apresenta dois métodos gerais — decomposição LU e decomposição LUP — para resolver equações lineares pelo método de eliminação de Gauss no tempo $O(n_3)$. Mostra também que inversão de matrizes e multiplicação de matrizes podem ser realizadas com rapidez igual. O capítulo termina mostrando como calcular uma solução aproximada de mínimos quadrados quando um conjunto de equações lineares não tem nenhuma solução exata.

O Capítulo 29 estuda programação linear, na qual desejamos maximizar ou minimizar um objetivo, dados recursos limitados e restrições concorrentes. A programação linear surge em uma variedade de áreas de aplicação prática. Esse capítulo abrange como formular e resolver programas lineares. O método de solução abordado é o algoritmo simplex, o mais antigo algoritmo para programação linear. Ao contrário de muitos algoritmos neste livro, o algoritmo simplex não é executado em tempo polinomial no pior caso, mas é razoavelmente eficiente e amplamente usado na prática.

O Capítulo 30 estuda operações com polinômios e mostra como usar uma técnica de processamento de sinais muito conhecida — a transformada rápida de Fourier (FFT)— para multiplicar dois polinômios de grau n no tempo $O(n \lg n)$. Investiga também implementações eficientes da FFT, incluindo um circuito paralelo.

O Capítulo 31 apresenta algoritmos da teoria dos números. Após revisar a teoria elementar dos números, apresenta o algoritmo de Euclides para calcular máximo divisor comum. Em seguida, estuda algoritmos para resolver equações lineares modulares e para elevar um número a uma potência módulo um outro número. Então, explora uma importante aplicação de algoritmos de teoria dos números: o criptossistema RSA de chaves públicas. O criptossistema pode ser usado não somente para criptografar mensagens de modo a impedir que um adversário as leia, mas também para fornecer assinaturas digitais. Então, o capítulo apresenta o teste de primalidade aleatorizado de Miller-Rabin, que nos permite encontrar eficientemente grandes números primos — um requisito essencial para o sistema RSA. Finalmente, o capítulo estuda a heurística “rô” de Pollard, para fatorar números inteiros e discute as técnicas mais modernas da fatoração de inteiros.

O Capítulo 32 estuda o problema de encontrar todas as ocorrências de uma dada cadeia-padrão em uma cadeia de texto dada, um problema que surge frequentemente em programas de edição de textos. Após examinar a abordagem ingênua, o capítulo apresenta uma abordagem elegante criada por Rabin e Karp. Então, depois de mostrar uma solução eficiente baseada em autômatos finitos, o capítulo apresenta o algoritmo de Knuth-Morris-Pratt, que modifica o algoritmo baseado em autômato para poupar espaço mediante o pré-processamento inteligente do padrão.

O Capítulo 33 considera alguns problemas de geometria computacional. Após discutir primitivas básicas da geometria computacional, o capítulo mostra como usar um método de "varredura" para determinar eficientemente se um conjunto de segmentos de reta contém quaisquer interseções. Dois algoritmos inteligentes para determinar a envoltória convexa de um conjunto de pontos — a varredura de Graham e a marcha de Jarvis — também ilustram o poder de métodos de varredura. O capítulo termina com um algoritmo eficiente para determinar o par mais próximo em um dado conjunto de pontos no plano.

O Capítulo 34 aborda problemas NP-completos. Muitos problemas computacionais interessantes são NP-completos, mas não há nenhum algoritmo de tempo polinomial conhecido para resolver qualquer deles. Esse capítulo apresenta técnicas para determinar quando um problema é NP-completo. Há vários problemas clássicos que são comprovadamente NP-completos: determinar se um grafo tem um ciclo hamiltoniano, determinar se uma fórmula booleana pode ser satisfeita e determinar se um determinado conjunto de números tem um subconjunto que equivale a um dado valor visado. O capítulo também prova que o famoso problema do caixeiro viajante é NP-completo.

O Capítulo 35 mostra como determinar eficientemente soluções aproximadas para problemas NP-completos usando algoritmos de aproximação. Para alguns problemas NP-completos é bastante fácil produzir soluções aproximadas quase ótimas, mas para outros, até mesmo os melhores algoritmos de aproximação conhecidos funcionam cada vez pior à medida que o tamanho do problema aumenta. Então, há alguns problemas para os quais podemos investir quantidades cada vez maiores de tempo de computação em troca de soluções aproximadas cada vez melhores. Esse capítulo ilustra tais possibilidades com o problema da cobertura de vértices (versões não ponderada e ponderada), uma versão de otimização de satisfazibilidade 3-CNF, o problema do caixeiro viajante, o problema da cobertura de conjuntos e o problema da soma de subconjuntos.

27 ALGORITMOS MULTITHREAD

A vasta maioria dos algoritmos neste livro são ***algoritmos seriais*** adequados para execução em um computador com um único processador no qual só uma instrução é executada por vez. Neste capítulo, estendemos nosso modelo algorítmico para abranger ***algoritmos paralelos***, que podem ser executados em computadores multiprocessadores que permitem a execução concorrente de várias instruções. Em particular, exploraremos o elegante modelo de algoritmos multithread dinâmicos, que se prestam ao projeto e à análise algorítmicos, bem como à implementação eficiente na prática. Computadores paralelos — computadores com várias unidades de processamento — são cada vez mais comuns e abrangem uma larga faixa de preços e desempenho. ***Chips multipro- cessadores*** relativamente baratos para computadores de mesa e laptops contêm um único chip de circuito integrado ***multinúcleo*** que abriga vários "núcleos" de processamento, cada um deles um processador totalmente desenvolvido que pode acessar uma memória comum. Em um ponto intermediário de preço/desempenho estão os grupos (clusters) montados com computadores individuais — quase sempre máquinas de classe PC — conectados por uma rede dedicada. As máquinas de preços mais altos são supercomputadores, que frequentemente usam uma combinação de arquiteturas customizadas e redes customizadas para oferecer o mais alto desempenho em termos de instruções executadas por segundo.

Computadores multiprocessadores já estão por aí, sob uma forma ou outra, há décadas. Embora a comunidade da computação tenha adotado o modelo da máquina de acesso aleatório para computação serial já no início da história da ciência da computação, nenhum modelo de computação paralela conquistou tão ampla aceitação. Uma razão principal é que os fabricantes não chegaram a um consenso quanto a um único modelo arquitetônico para computadores paralelos. Por exemplo, alguns computadores paralelos possuem ***memória compartilhada***, na qual cada processador pode acessar diretamente qualquer localização da memória. Outros computadores paralelos empregam ***memória distribuída***, na qual a memória de cada processador é particular e é preciso enviar uma mensagem explícita entre processadores para que um processador acesse a memória de outro. Porém, com o advento da tecnologia multinúcleo, agora cada novo laptop e computador de mesa é um computador paralelo de memória compartilhada, e aparentemente a tendência é a favor do multiprocessamento por memória compartilhada. Embora somente o tempo dirá, essa é a abordagem que adotaremos neste capítulo.

Um meio comum de programar chips multiprocessadores e outros computadores paralelos de memória compartilhada é usar ***thread estático***, que proporciona uma abstração em software de linhas de execução ou ***threads****, que compartilham uma memória comum. Cada thread mantém um contador de programa associado e pode executar código independentemente dos outros threads. O sistema operacional carrega um thread em um processador para execução e o desliga quando um outro thread precisa ser executado. Embora o sistema operacional permita que programadores criem ou destruam threads, essas operações são comparativamente lentas. Assim, para a maioria das aplicações, os threads persistem durante o tempo de uma computação, e é por essa razão que são denominados "estáticos".

Infelizmente, programar diretamente um computador paralelo de memória compartilhada usando threads estáticos é difícil e sujeito a erro. Uma razão é que repartir o trabalho dinamicamente entre os threads de modo que cada um receba aproximadamente a mesma carga se revela uma empreitada complicada. Para quaisquer aplicações, exceto as mais simples, o programador deve usar protocolos de comunicação complexos para implementar um escalonador para

equilibrar a carga de trabalho. Esse estado de coisas levou à criação de ***plataformas de concorrência***, que oferecem uma camada de software que coordena, escalona e gerencia os recursos de computação paralela. Algumas plataformas de concorrência são construídas como bibliotecas de tempo real, mas outras oferecem linguagens paralelas totalmente desenvolvidas com compilador e suporte em tempo real.

Programação com multithread dinâmico

Uma classe importante de plataforma de concorrência é o ***multithread dinâmico***, modelo que adotaremos neste capítulo. Multithread dinâmico permite que programadores especifiquem paralelismo em aplicações sem que tenham de se preocupar com protocolos de comunicação, balanceamento de carga e outros eventos imprevisíveis da programação com threads estáticos. A plataforma de concorrência contém um escalonador que equilibra a carga de computação automaticamente, o que simplifica muito a tarefa do programador. Embora a funcionalidade dos ambientes de multithread dinâmico ainda esteja em desenvolvimento, quase todos suportam duas características: paralelismo aninhado e laços paralelos. Paralelismo aninhado permite que uma sub-rotina seja "gerada" (*spawned*), permitindo que o chamador prossiga enquanto a sub-rotina gerada está calculando o seu resultado. Um laço paralelo é como um laço **for** comum, exceto que as iterações do laço podem ser executadas concorrentemente.

Essas duas características formam a base do modelo de multithread dinâmico que estudaremos neste capítulo. Um aspecto fundamental desse modelo é que o programador precisa especificar somente o paralelismo lógico dentro de uma computação, e os threads dentro da plataforma de concorrência subjacente escalonam e equilibram a carga da computação entre eles. Investigaremos algoritmos multithread escritos para esse modelo, e também como a plataforma de concorrência subjacente pode escalonar computações eficientemente.

Nosso modelo para multithread dinâmico oferece várias vantagens importantes:

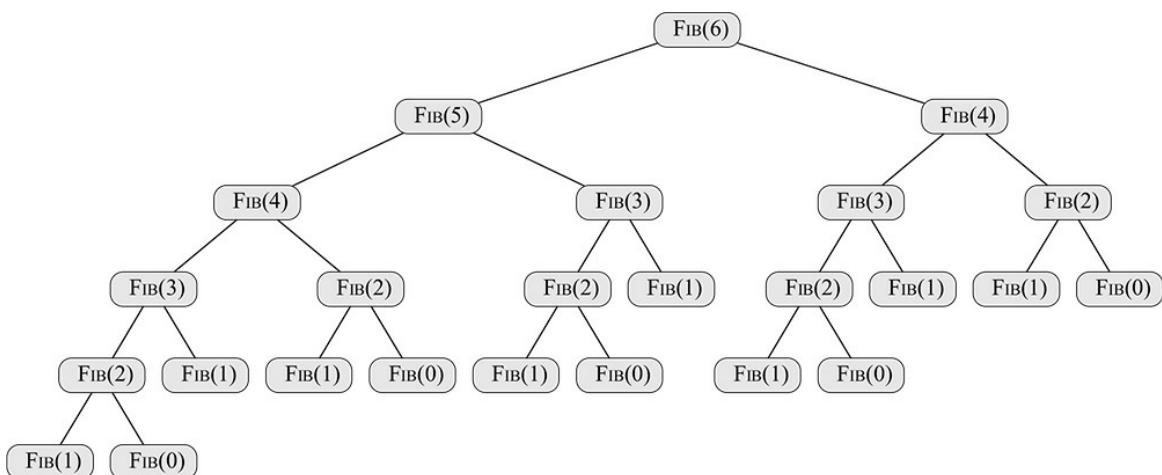
- É uma simples extensão do nosso modelo de programação serial. Podemos descrever um algoritmo multithread acrescentando ao nosso pseudocódigo apenas três palavras-chaves de "concorrência": **parallel**, **spawn** e **sync**. Além do mais, se eliminarmos essas palavras-chaves de concorrência do pseudocódigo multithread, o texto resultante é pseudocódigo serial para o mesmo problema, o que denominamos "serialização" do algoritmo multithread.
- Proporciona um modo teoricamente claro para quantificar paralelismo com base nas noções de "trabalho" e "duração".
- Muitos algoritmos multithread que envolvem paralelismo aninhado decorrem naturalmente do paradigma divisão e conquista. Além do mais, exatamente como algoritmos seriais de divisão e conquista se prestam à análise por solução de recorrências, também os algoritmos multithread se prestam a essa mesma análise.
- O modelo é fiel ao modo de evolução da prática da computação paralela. Um número cada vez maior de plataformas de concorrência suportam uma ou outra variante do multithread dinâmico, entre elas Cilk [51, 118], Cilk++ [72], OpenMP [60], Task Paralellel Library [230] e Threading Building Blocks [292].

A Seção 27.1 apresenta o modelo de multithread dinâmico e as métricas de trabalho, duração e paralelismo, que usaremos para analisar algoritmos multithread. A Seção 27.2 investiga como multiplicar matrizes com multithread, e a Seção 27.3 ataca o problema mais árduo de aplicar multithread à ordenação por inserção.

27.1 OS FUNDAMENTOS DO MULTITHREAD DINÂMICO

Começaremos nossa exploração do multithread dinâmico usando o exemplo do cálculo recursivo de números de Fibonacci. Lembre-se de que números de Fibonacci são definidos por recorrência (3.22):

$F_0 = 0$,
$F_1 = 1$,
$F_i = F_{i-1} + F_{i-2}$ para $i \geq 2$.

Apresentamos a seguir um algoritmo serial simples, recursivo, para calcular o n-ésimo número de Fibonacci:

```
FIB(n)
1      if n ≤ 1
2          return n
3      else x = FIB(n − 1)
4          y = FIB(n − 2)
5          return x + y
```

Na realidade não seria interessante calcular grandes números de Fibonacci desse modo, porque esse cálculo realiza muito trabalho repetido. A Figura 27.1 mostra a árvore de instâncias de procedimento recursivo que são criadas no cálculo de F_6. Por exemplo, uma chamada a FIB(6) chama recursivamente FIB(5) e então FIB(4). Mas a chamada a FIB(5) também resulta em uma chamada a FIB(4). Ambas as instâncias de FIB(4) retornam o mesmo resultado ($F_4 = 3$). Visto que o procedimento FIB não memoíza, a segunda chamada a FIB(4) repete o trabalho que a primeira chamada realiza.

Seja $T(n)$ o tempo de execução de FIB(n). Visto que FIB(n) contém duas chamadas recursivas mais uma quantidade constante de trabalho extra, obtemos a recorrência

$$T(n) = T(n-1) + T(n-2) + \Theta(1):$$

Essa recorrência tem solução $T(n) = (F_n)$, o que podemos mostrar utilizando o método de substituição. Para uma hipótese de indução, suponha que $T(n) \leq a\, F_n - b$, onde $a > 1$ e $b > 0$ são constantes. Substituindo, obtemos

Figura 27.1 A árvore de instâncias de procedimento recursivo para o cálculo de FIB(6). Cada instância de FIB que tenha os mesmos argumentos realiza o mesmo trabalho para produzir o mesmo resultado, proporcionando um modo ineficiente, mas interessante de calcular números de Fibonacci.

$$\begin{aligned} T(n) &\leq (a\, F_{n-1} - b) + (a\, F_{n-2} - b) + \Theta(1) \\ &= a(F_{n-1} + F_{n-2}) - 2b + \Theta(1) \\ &= a\, F_n - b - (b - \Theta(1)) \\ &\leq a\, F_n - b \end{aligned}$$

se escolhermos b grande o suficiente para dominar a constante no (1). Então podemos escolher a grande o suficiente para satisfazer a condição inicial. O limite analítico

$$T(n) = \Theta(\phi^n), \qquad (27.1)$$

onde = $(1 + \sqrt{5})/2$ é a razão áurea, decorre agora da equação (3.25).

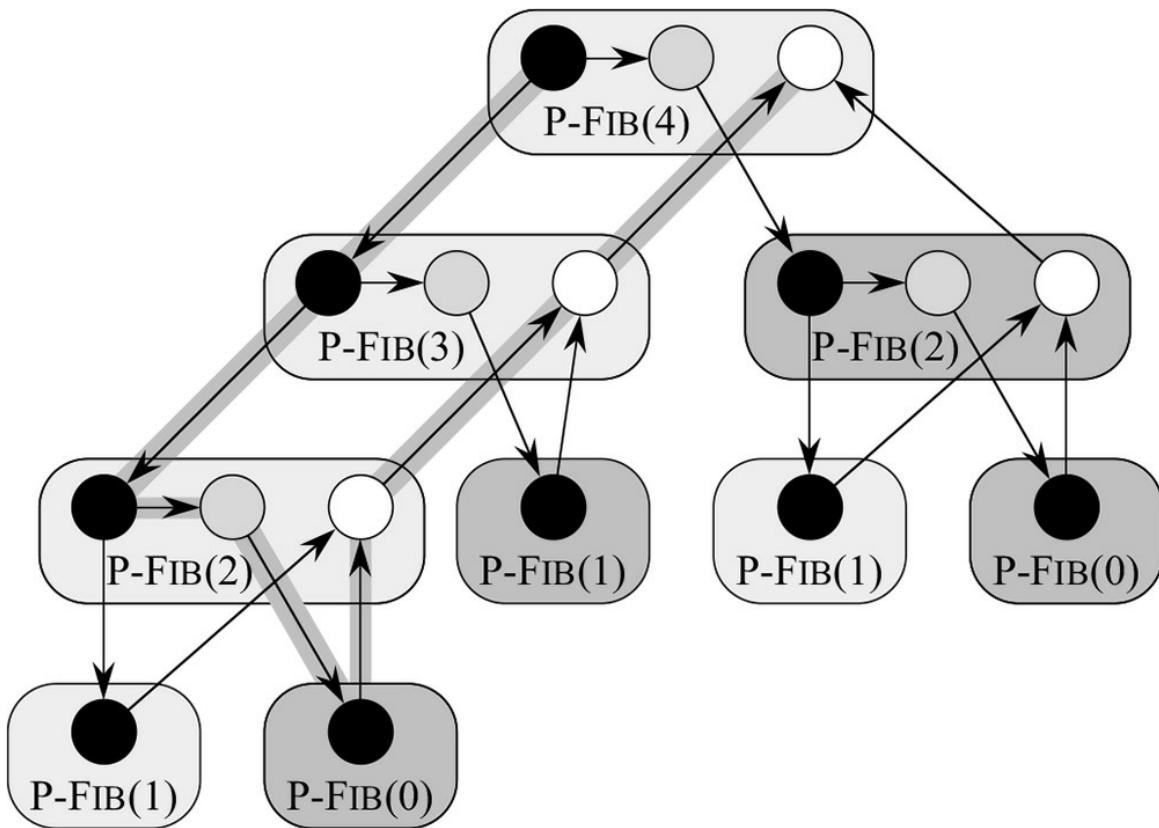
Visto que F_n cresce exponencialmente em n, esse procedimento é um modo particularmente lento de calcular números de Fibonacci. (Veja modos mais rápidos no Problema 31-3.)

Embora seja um modo ruim de calcular números de Fibonacci, o procedimento FIB constitui um bom exemplo para ilustrar os principais conceitos na análise de algoritmos multithread. Observe que dentro de FIB(n), as duas chamadas recursivas nas linhas 3 e 4 a FIB(n–1) e FIB(n–2), respectivamente, são independentes uma da outra: poderiam ser chamadas em qualquer ordem, e a computação executada por uma delas em nada afeta a outra. Portanto, as duas chamadas recursivas podem ser executadas em paralelo.

Aumentamos nosso pseudocódigo para indicar paralelismo acrescentando as ***palavras-chaves de concorrência*** **spawn** e **sync**. Agora, apresentamos como podemos reescrever o procedimento FIB para usar multithread dinâmico:

P-FIB(n)
1 **if** $n \leq 1$
2 **return** n
3 **else** $x =$ **spawn** P-FIB($n - 1$)
4 $y =$ P-FIB($n - 2$)
5 **sync**
6 **return** $x + y$

Observe que, se eliminarmos as palavras-chaves de concorrência **spawn** e **sync** de P-FIB, o texto de pseudocódigo resultante é idêntico ao de FIB (exceto o nome do procedimento no cabeçalho e nas duas chamadas recursivas). Definimos a ***serialização*** de um algoritmo multithread como o algoritmo serial que resulta da eliminação das palavras-chaves multithread: **spawn**, **sync** e, quando examinamos laços, **parallel**. De fato, nosso pseudocódigo multithread tem a seguinte propriedade atraente: uma serialização é sempre pseudocódigo serial comum para resolver o mesmo problema.

Paralelismo aninhado ocorre quando a palavra-chave **spawn** precede uma chamada de procedimento, como na linha 3. A semântica de uma geração (*spawn*) é diferente da semântica de uma chamada de procedimento comum, no sentido de que a instância de procedimento que executa a *geração* — o ***pai*** — pode continuar a ser executada em paralelo com a sub-rotina gerada — seu ***filho***, em vez de esperar pela conclusão do filho, como normalmente aconteceria em uma execução serial. Nesse caso, enquanto o filho gerado está calculando P-FIB(n – 1), o pai pode continuar a calcular P-FIB(n – 2) na linha 4 em paralelo com o filho gerado. Visto que o procedimento P-FIB é recursivo, essas mesmas duas chamadas de sub-rotinas criam paralelismo alinhado, assim como seus filhos, criando assim uma árvore potencialmente vasta de subcomputações, todas executadas em paralelo.

Contudo, a palavra-chave **spawn** não diz que um procedimento *deve* ser executado concorrentemente com seu filho gerado, diz apenas que *pode.* As palavras-chaves de concorrência expressam o ***paralelismo lógico*** da computação, indicando quais partes da computação podem prosseguir em paralelo. Em tempo real, cabe a um ***escalonador*** determinar quais subcomputações realmente são executadas concorrentemente designando a elas processadores disponíveis à medida que a computação transcorre. Em breve, discutiremos a teoria que fundamenta escalonadores.

Um procedimento não pode utilizar com segurança os valores retornados por seus filhos gerados até executar uma declaração **sync**, como na linha 5. A palavra-chave **sync** indica que o procedimento deve esperar o quanto for necessário pela conclusão de todos os seus filhos gerados antes de passar para a declaração depois de **sync**. No procedimento P-FIB, é preciso uma **sync** antes da declaração **return** na linha 6 para evitar a anomalia que ocorreria se x e y fossem somados antes de x ser calculado. Além da sincronização explícita proporcionada pela declaração **sync**,

todo procedimento executa uma **sync** implicitamente antes de retornar, garantindo assim que todos os seus filhos terminam antes dela.

Um modelo para execução multithread

Ajuda imaginar uma ***computação multithread*** — o conjunto de instruções em tempo real executadas por um processador em nome de um programa multithread — como um grafo acíclico dirigido $G = (V, E)$, denominado um ***gad de computação.*** Como exemplo, a Figura 27.2 mostra o gad de computação que resulta do cálculo de P-FIB(4). Conceitualmente, os vértices em V são instruções, e as arestas em E representam dependências entre instruções, onde $(u, v) \in E$ significa que a instrução u deve ser executada antes da instrução v. Todavia, por conveniência, se uma cadeia de instruções não contém nenhum controle paralelo (nenhuma **spawn**, **sync** ou **return** de uma **spawn** — seja por meio de uma declaração **return** explícita ou por meio de um retorno que ocorre implicitamente ao chegar ao fim de um procedimento), podemos agrupá-las em uma única **fibra** (***strand)***, cada uma das quais representa uma ou mais instruções. Instruções que envolvem controle paralelo não são incluídas em fibras, mas são representadas na estrutura do gad. Por exemplo, se uma fibra tem dois sucessores, um deles deve ter sido gerado, e uma fibra com vários predecessores indica que os predecessores se uniram por causa de uma declaração **sync.** Assim, no caso geral, o conjunto V forma o conjunto de fibras, e o conjunto E de arestas dirigidas, representa dependências entre fibras induzidas por controle paralelo. Se G tem um caminho dirigido da fibra u até a fibra v, dizemos que as duas fibras estão ***(logicamente) em série***. Caso contrário, os filamentos u e v estão ***(logicamente) em paralelo***.

Figura 27.2 Um grafo acíclico dirigido que representa a computação de P-FIB(4). Cada círculo representa uma fibra. Círculos pretos representam ou casos bases ou parte do procedimento (instância) até a geração do P-FIB (n - 1) na linha 3. Círculos sombreados representam a parte do procedimento que chama P-FIB(n - 2) na linha 4 até **sync** na linha 5, onde fica suspenso até que P-FIB(n - 1) retorne. Círculos brancos representam a parte do procedimento depois de **sync** onde ele soma x e y até o ponto em que retorna o resultado. Cada grupo de fibras que pertencem ao mesmo procedimento é cercada por um retângulo de cantos arredondados, sombreado em tom mais claro para procedimentos gerados e em tom mais escuro para procedimentos chamados. Em arestas geradas e arestas de chamada as setas apontam para baixo, em arestas de continuação elas apontam para a direita na horizontal e em arestas de

retorno apontam para cima. Considerando que cada fibra demora uma unidade de tempo, o trabalho é igual a 17 unidades de tempo, visto que há 17 filamentos, e a duração é 8 unidades de tempo, já que o caminho crítico — mostrado por arestas sombreadas — contém 8 fibras.

Podemos representar uma computação multithread como um gad de fibras embutido em uma árvore de instância de procedimentos. Por exemplo, a Figura 27.1 mostra a árvore de instâncias de procedimento para P-FIB(6) sem a estrutura detalhada que mostra as fibras. A Figura 27.2 mostra o detalhe de uma seção daquela árvore, onde podemos ver as fibras que constituem cada procedimento. Todas as arestas dirigidas que conectam fibras são executadas ou dentro de um procedimento ou ao longo de arestas não dirigidas na árvore de procedimentos.

Podemos classificar as arestas de um gad de computação para indicar o tipo de dependências entre as várias fibras. Uma ***aresta de continuação*** (u, u'), desenhada na horizontal na Figura 27.2, conecta uma fibra u a seu sucessor u' dentro da mesma instância de procedimento. Quando uma fibra u gera uma fibra v, um gad contém uma a ***aresta geradora*** (u, v), que aponta para baixo na figura. ***Arestas de chamada***, que representam chamadas de procedimento normais, também apontam para baixo. Fibra u gerar fibra v é diferente de u chamar v no sentido de que uma geração induz uma aresta de continuação horizontal de u até a fibra u' e segue u em seu procedimento, indicando que u' está livre para ser executado ao mesmo tempo que v, ao passo que uma chamada não induz tal aresta. Quando uma fibra u retorna a seu procedimento de chamada e x é o filamento que vem imediatamente após a próxima **sync** no procedimento de chamada, o gad de computação contém a ***aresta de retorno*** (u, x), que aponta para cima. Uma computação começa com uma única ***fibra inicial*** — o vértice preto no procedimento identificado por P-FIB(4) na Figura 27.2—e termina com uma única ***fibra final*** — o vértice branco no procedimento identificado por P-FIB(4).

Estudaremos a execução de algoritmos multithread em um ***computador paralelo ideal***, que consiste em um conjunto de processadores e uma memória compartilhada ***sequencialmente con- sistente***. Consistência sequencial significa que a memória compartilhada, que na realidade pode estar executando muitas cargas e armazenamentos de processadores ao mesmo tempo, produz os mesmos resultados como se a cada etapa executasse exatamente uma instrução de um dos processadores. Isto é, a memória se comporta como se as instruções fossem executadas sequencialmente de acordo com alguma ordem linear global que preserva as ordens individuais nas quais cada processador emite suas próprias instruções. Para computações multithread dinâmicas, que são escalonadas para processadores automaticamente pela plataforma de concorrência, a memória compartilhada comporta-se como se as instruções da computação multithread fossem intercaladas para produzir uma ordem linear que preserva a ordem parcial do gad de computação. Dependendo do escalonamento, a ordenação pode ser diferente de uma execução do programa para outra, mas o comportamento de qualquer execução pode ser entendido considerando que as instruções dão executadas em alguma ordem linear consistente com o gad de computação.

Além de tais considerações semânticas, o modelo do computador paralelo ideal também faz algumas considerações de desempenho. Especificamente, o modelo supõe que cada processador na máquina tem a mesma capacidade de computação e ignora o custo do escalonamento. Embora essa última suposição possa parecer otimista, acontece que para algoritmos com "paralelismo" suficiente (um termo que definiremos exatamente em breve), a sobrecarga de escalonamento geralmente é mínima na prática.

Medidas de desempenho

Podemos aferir a eficiência teórica de um algoritmo multithread medindo duas medidas: "trabalho" e "duração." O ***trabalho*** de uma computação multithread é o tempo total para executar a computação inteira em um processador. Em outras palavras, o trabalho é a soma dos tempos gastos por cada uma das fibras. Para um gad de computação no qual cada fibra demora um tempo unitário, o trabalho é apenas o número de vértices no gad. A ***duração*** é o tempo mais longo para executar as fibras ao longo de qualquer caminho no gad.

Novamente, para um gad no qual cada fibra demora um tempo unitário, a duração é igual ao número de vértices em um caminho de comprimento máximo, ou ***caminho crítico*** no gad. (Lembre-se de que a Seção 24.2 mostrou que

podemos encontrar um caminho crítico em um gad $G = (V, E)$ no tempo $(V + E)$). Por exemplo, o gad de computação da Figura 27.2 tem 17 vértices ao todo e 8 vértices em seu caminho crítico, de modo que, se cada fibra demorar tempo unitário, seu trabalho é 17 unidades de tempo e sua duração é 8 unidades de tempo.

O tempo de execução propriamente dito de uma computação multithread depende não somente de seu trabalho e de sua duração, mas também de quantos processadores estão disponíveis e de como o escalonador aloca fibras a processadores. Para denotar o tempo de execução de uma computação multithread em P processadores, usaremos o subscrito P. Por exemplo, poderíamos denotar o tempo de execução de um algoritmo em P processadores por T_P . O trabalho é o tempo de execução em um único processador, ou T_1. A duração é o tempo de execução se pudéssemos executar cada fibra em seu próprio processador—em outras palavras, se tivéssemos um número ilimitado de processadores — e portanto, denotamos a duração por T_∞.

O trabalho e a duração dão limites inferiores para o tempo de execução T_P de uma computação multithread em P processadores:

- Em uma etapa, um computador paralelo ideal com P processadores pode realizar no máximo P unidades de trabalho, e assim, no tempo T_P, ele pode realizar máximo o trabalho PT_P. Visto que o trabalho total a realizar é T_1, temos $PT_P \geq T_1$. Dividindo por P obtemos a ***lei do trabalho***:

$$T_P \geq T_1/P. \tag{27.2}$$

- Um computador paralelo ideal com P processadores não pode executar mais rapidamente do que uma máquina com um número ilimitado de processadores. Visto por outro ângulo, uma máquina com um número ilimitado de processadores pode emular uma máquina com P processadores utilizando apenas P de seus processadores. Assim, decorre a ***lei da duração***:

$$T_P \geq T_\infty. \tag{27.3}$$

Definimos o ***fator de aceleração*** de uma computação em P processadores pela razão T_1/T_P, que diz quantas vezes uma computação com P processadores é mais rápida que com 1 processador. Pela lei do trabalho, temos $T_P \geq T_1/P$, o que implica que $T_1/T_P \leq P$. Assim, o fator de aceleração em P processadores pode ser no máximo P. Quando o fator de aceleração é linear em relação ao número de processadores, isto é, quando $T_1/T_P = (P)$, a computação exibe ***fator de aceleração linear***, e quando $T_1/T_P = P$ temos ***fator de aceleração linear perfeito.***

A razão T_1/T_∞ entre o trabalho e a duração dá o ***paralelismo*** da computação multithread. Podemos enxergar o paralelismo de três pontos de vista. Como uma razão, o paralelismo denota a quantidade média de trabalho que pode ser realizada em paralelo para cada etapa ao longo do caminho crítico. Como um limite superior, o paralelismo dá o máximo fator de aceleração possível que pode ser conseguido com qualquer número de processadores. Finalmente, e talvez mais importante, o paralelismo dá um limite para a possibilidade de conseguir fator de aceleração linear perfeito. Especificamente, tão logo o número de processadores exceda o paralelismo, não há nenhuma possibilidade de conseguir fator de aceleração linear perfeito na computação. Para ver esse último ponto, suponha que $P > T_1/T_\infty$, caso em que a lei da duração implica que o fator de aceleração satisfaz $T_1/T_P \leq T_1/T_\infty < P$. Além disso, se o número P de processadores no computador paralelo ideal for muito maior do que o paralelismo — isto é, se $P >> T_1/T_\infty$ — então $T_1/T_P << P$, de modo que o fator de aceleração é muito menor que o número de processadores. Em outras palavras, quanto mais processadores usarmos além do paralelismo, menos perfeito o fator de aceleração.

Como exemplo, examine a computação P-FIB(4) na Figura 27.2, e suponha que cada fibra demora tempo unitário. Visto que o trabalho é $T_1 = 17$ e a duração é $T_\infty = 8$, o paralelismo é $T_1/T_\infty = 17/8 = 2{,}125$. Consequentemente, conseguir muito mais que o dobro do fator de aceleração é impossível, não importando quantos processadores empregamos para executar a computação. Contudo, para tamanhos maiores de entradas, veremos que P-FIB(n) exibe substancial paralelismo.

Definimos a ***folga (paralela)*** de uma computação multithread executada em um computador paralelo ideal com P processadores como a razão $(T_1/T_\infty)/P = T_1/(PT_\infty)$, que é o fator que representa quantas vezes o paralelismo da

computação excede o número de processadores na máquina. Assim, se a folga é menor que 1, não podemos esperar conseguir um fator de aceleração linear perfeito, porque $T_1/(PT_\infty) < 1$ e a lei da duração implica que o fator de aceleração em P processadores satisfaz $T_1/T_P \leq T_1/T_\infty < P$. Realmente, à medida que folga diminui de 1 até 0, o fator de aceleração da computação diverge cada vez mais do fator de aceleração linear perfeyto. Todavia, se a folga é maior que 1, o trabalho por processador é a restrição limitativa. Como veremos, à medida que a folga aumenta em relação a 1, um bom escalonador pode chegar cada vez mais perto de um fator de aceleração linear perfeito.

Escalonamento

Bom desempenho depende de mais coisas do que apenas minimizar trabalho e duração. As fibras também têm de ser escalonadas eficientemente nos processadores da máquina paralela. Nosso modelo de programação multithread não dá nenhum modo para especificar quais fibras executar em quais processadores. Em vez disso, confiamos no escalonador da plataforma de concorrência para mapear a computação que está se desenvolvendo dinamicamente para processadores individuais. Na prática, o escalonador mapeia as fibras para threads estáticos e o sistema operacional escalona os threads nos próprios escalonadores, mas esse nível extra de indireção é desnecessário para entendermos o escalonamento. Basta imaginar que o escalonador da plataforma de concorrência mapeia fibras para processadores diretamente.

Um escalonador multithread deve escalonar a computação sem saber de antemão quando as fibras serão geradas ou quando serão concluídos — ele tem de funcionar ***on-line***. Além do mais, um bom escalonador funciona de modo distribuído, no qual os threads que implementam o escalonador cooperam no equilíbrio de cargas da computação. Existem escalonadores distribuídos, on-line provadamente bons, mas analisá-los é complicado.

Em vez disso, para manter a simplicidade de nossa análise, investigaremos um escalonador ***centralizado*** on-line, que sabe qual é o estado global da computação a qualquer tempo dado. Em particular, analisaremos ***escalonadores gulosos***, que designam o maior número possível de filamentos a processadores em cada etapa de tempo. Se no mínimo P fibras estão prontas para executar durante uma etapa de tempo, dizemos que essa é uma ***etapa completa***, e um escalonador guloso designa qualquer P das fibras prontas a processadores. Caso contrário, um número menor do que P fibras estão prontas para executar, e então dizemos que essa é uma ***etapa incompleta***, e o escalonador designa cada fibra pronta a seu próprio processador.

Pela lei do trabalho, o melhor tempo de execução que podemos esperar para P processadores é $T_P = T_1/P$, e pela lei da duração o melhor que podemos esperar é $T_P = T$. O teorema a seguir mostra que o escalonamento guloso é provadamente bom no sentido de que faz da soma desses dois limites inferiores um limite superior.

Teorema 27.1

Em um computador paralelo ideal com P processadores, um escalonador guloso executa uma computação multithread com trabalho T_1 e duração T_∞ no tempo

$$T_P \leq T_1/P + T_\infty. \qquad (27.4)$$

Prova Começamos considerando as etapas completas. Em cada etapa completa, os P processadores juntos executam um total de trabalho P. Suponha, por contradição, que o número de etapas completas é estritamente maior que T_1/P.

Então, o trabalho total das etapas completas é no mínimo

$$\begin{aligned} P \cdot (\lfloor T_1/P \rfloor + 1) &= P\lfloor T_1/P \rfloor + P \\ &= T_1 - (T_1 \bmod P) + P && \text{(pela equação (3.8))} \\ &> T_1 && \text{(pela desigualdade (3.9))}. \end{aligned}$$

Assim, obtemos a contradição que os P processadores executariam mais trabalho que o exigido pela computação, o que nos permite concluir que o número de etapas completas é no máximo T_1/P .

Agora, considere uma etapa incompleta. Seja G o gad que representa a computação inteira e, sem perda da generalidade, suponha que cada filamento demora tempo unitário. (Podemos substituir cada fibra mais longa por uma cadeia de fibras de tempo unitário.) Seja G' o subgrafo de G que ainda tem de ser executado no início da etapa incompleta, e seja G'' o subgrafo que resta para ser executado depois da etapa incompleta. Um caminho de comprimento máximo em um gad deve necessariamente começar em um vértice com grau de entrada 0. Visto que uma etapa incompleta de um escalonador guloso executa todas as fibras com grau de entrada 0 em G', o comprimento do caminho de comprimento máximo em G'' deve ter uma unidade a menos que o comprimento do caminho de comprimento máximo em G'. Em outras palavras, uma etapa incompleta diminui de 1 a duração do gad não executado. Por consequência, o número de etapas incompletas é no máximo T_∞.
Visto que cada etapa ou é complete ou é incompleta, o teorema segue.

O seguinte corolário para o Teorema 27.1 mostra que um escalonador guloso sempre funciona bem.

Corolário 27.2

O tempo de execução T_P de qualquer computação multithread escalonada por um escalonador guloso em um computador paralelo ideal com P processadores está dentro de um fator de 2 em relação ao ótimo.

Prova Seja T_*P o tempo de execução produzido por um escalonador ótimo em uma máquina com P processadores, e sejam T_1 e T_∞ o trabalho e a duração da computação, respectivamente. Visto que as leis do trabalho e da duração — desigualdades (27.2) e (27.3) — nos dão $T^*{}_P \geq \max(T_1/P, T_\infty)$, o Teorema 27.1 implica que

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2 \cdot \max(T_1/P, T_\infty) \\ &\leq 2\,T_P^*. \end{aligned}$$

O próximo corolário mostra que, de fato, um escalonador guloso consegue fator de aceleração linear quase perfeito em qualquer computação multithread à medida que a folga aumenta.

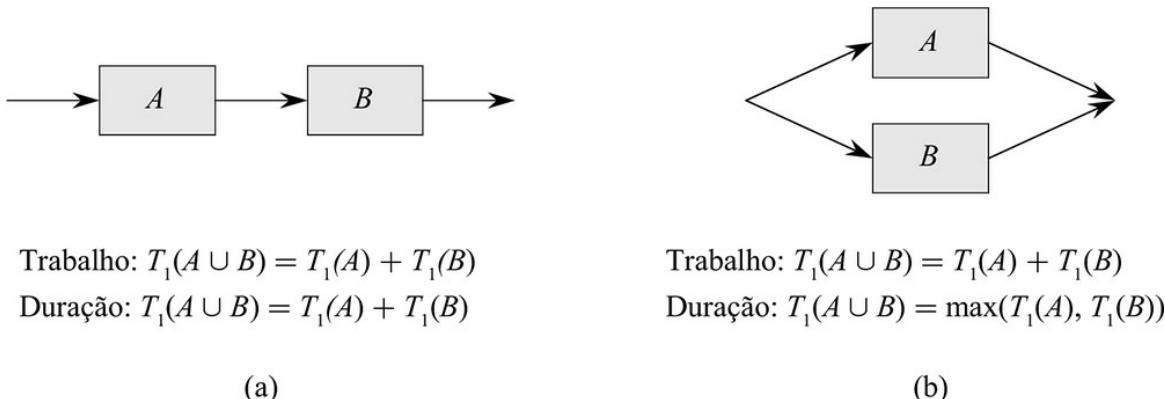
Corolário 27.3

Seja T_P o tempo de execução de uma computação multithread produzida por um escalonador guloso em um computador paralelo ideal com P processadores, e sejam T_1 e T_∞ o trabalho e a duração da computação, respectivamente. Então, se $P << T_1/T_\infty$, temos $T_P \approx T_1/P$, ou o que é equivalente, um fator de aceleração de aproximadamente P.

Prova Se supusermos que $P << T_1/T_\infty$, então temos também $T_\infty << T_1/P$ e, por consequência, o Teorema 27.1 nos dá $T_P \leq T_1/P + T_\infty \approx T_1/P$. Visto que a lei do trabalho (27.2) impõe que $T_P \geq T_1/P$, concluímos que $T_P \approx T_1/P$, ou o que é equivalente, que o fator de aceleração é $T_1/T_P \approx P$.

O símbolo << denota "muito menos," mas quanto menos é "muito menos"? Como regra prática, uma folga de no mínimo 10 — isto é, 10 vezes mais paralelismo que processadores — geralmente é suficiente para conseguir um bom fator de aceleração. Então, o termo de duração no limite guloso, desigualdade (27.4), é menor que 10% do tempo de trabalho por processador, que é suficientemente bom para a maioria das situações encontradas na engenharia. Por exemplo, se uma computação for executada em somente 10 ou 100 processadores, não tem sentido preferir paralelismo de, digamos, 1.000.000 em comparação com o paralelismo de 10.000, mesmo com o fator de 100 para a diferença. Como mostra o Problema 27-2, reduzindo o paralelismo extremo, às vezes podemos obter algoritmos que são melhores em relação a outros tópicos e que aproveitam bem um número razoável de processadores.

Análise de algoritmos multithread

Agora, temos todas as ferramentas que precisamos para analisar algoritmos multithread e conseguir bons limites para seus tempos de execução com vários números de processadores. Analisar o trabalho é relativamente direto, visto que nada mais é do que analisar o tempo de execução de um algoritmo serial comum — isto é, a serialização do algoritmo multithread — com o que você já deve estar familiarizado, visto que a quase totalidade deste livro trata dele! Analisar a duração é mais interessante, porém, em geral, não é mais difícil, desde que você pegue o jeito da coisa. Investigaremos as idéias básicas utilizando o programa P-FIB.

Analisar o trabalho $T_1(n)$ de P-FIB(n) não apresenta obstáculos, porque já o fizemos. O procedimento FIB original é essencialmente a serialização de P-FIB e, por consequência, $T_1(n) = T(n) = (_n)$ pela equação (27.1).

A Figura 27.3 ilustra como analisar a duração. Se duas subcomputações estão ligadas em série, suas durações se somam para formar a duração de sua composição, ao passo que, se estiverem ligadas em paralelo, a duração de sua composição é o máximo das durações das duas subcomputações. Para P-FIB(n), a chamada gerada a P-FIB(n–1) na linha 3 é executada em paralelo com a chamada a P-FIB(n – 2) na linha 4. Por consequência, podemos expressar a duração de P-FIB(n) como a recorrência

$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1), \end{aligned}$$

cuja solução é $T_\infty(n) = (n)$.

O paralelismo de P-FIB(n) é $T_1(n)/T_\infty(n) = (/n)$, que cresce dramaticamente à medida que n fica grande. Assim, mesmo nos maiores computadores paralelos, um valor modesto para n é suficiente para conseguir um fator de aceleração linear quase perfeito para P-FIB(n), porque esse procedimento exibe folgas paralelas consideráveis.

Figura 27.3 O trabalho e a duração de subcomputações compostas. **(a)** Quando duas subcomputações são ligadas em série, o trabalho da composição é a soma de seus trabalhos, e a duração da composição pela soma de suas durações. **(b)** Quando duas subcomputações são ligadas em paralelo, o trabalho de composição continua sendo a soma de seus trabalhos, mas a duração da composição é somente o máximo de suas durações.

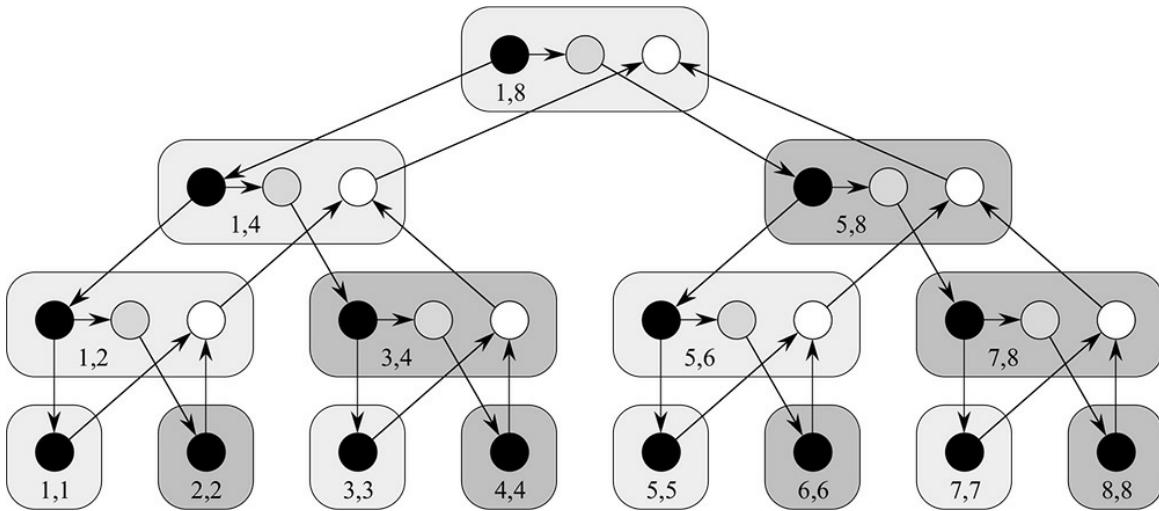
Laços paralelos

Muitos algoritmos contêm laços nos quais todas as iterações podem funcionar em paralelo. Como veremos, podemos paralelizar tais laços utilizando as palavras-chave **spawn** e **sync**, porém é muito mais conveniente especificar diretamente que as iterações de tais laços podem ser executadas concorrentemente. Nosso pseudocódigo proporciona essa funcionalidade por meio da palavra-chave de concorrência **parallel**, que precede a palavra-chave **for** em uma declaração de laço **for**.

Como exemplo, considere o problema de multiplicar uma matriz $n \times n$ $A = (a_{ij})$ por um n-vetor $x = (x_j)$. O n-vetor resultante $y = (y_i)$ é dado pela equação

$$y_i = \sum_{j=1}^{n} a_{ij} x_j ,$$

para $i = 1, 2, \dots, n$. Podemos efetuar a multiplicação matriz-vetor calculando todas as entradas de y em paralelo da seguinte maneira:

```
MAT-VEC(A, x)
1    n = A.linhas
2    seja y um novo vetor de comprimento n
3    parallel for i = 1 to n
4         y_i = 0
5    parallel for i = 1 to n
6         for j = 1 to n
7              y_i = y_i + a_ij x_j
8    return y
```

Nesse código, as palavras-chave **parallel for** nas linhas 3 e 5 indicam que as iterações dos respectivos laços podem ser executadas concorrentemente. Um compilador pode implementar cada laço **parallel for** como uma sub-rotina com divisão e conquista utilizando paralelismo aninhado. Por exemplo, o laço **parallel for** nas linhas 5–7 pode ser implementado com a chamada MAT-VEC-MAIN-LOOP(A, x, y, n, 1, n), onde o compilador produz a sub-rotina auxiliar MAT-VEC- -MAIN-LOOP da seguinte maneira:

```
MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')
1    if i == i'
2         for j = 1 to n
3              y_i = y_i + a_ij x_j
4    else mid = ⌊(i + i')/2⌋
5         spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
6         MAT-VEC-MAIN-LOOP(A, x, y, n, mid + 1, i')
7         sync
```

Esse código gera recursivamente a primeira metade das iterações do laço para ser executada em paralelo com a segunda metade das iterações e depois executa uma **sync**, criando assim uma árvore binária de execução na qual as folhas são iterações de laço individuais, como mostra a Figura 27.4.

Para calcular o trabalho $T_1(n)$ de MAT-VEC em uma matriz $n \times n$, simplesmente calculamos o tempo de execução de sua serialização, o que obtemos substituindo os laços **parallel for** por laços **for** comuns. Assim, temos $T_1(n) = (n_2)$, porque o tempo de execução quadrático dos laços duplamente aninhados nas linhas 5-7 domina. Contudo, essa análise parece ignorar a sobrecarga para geração recursiva na implementação dos laços paralelos. Na verdade, a sobrecarga da geração recursiva aumenta o trabalho de um laço paralelo em comparação com o trabalho de sua serialização, mas não assintoticamente. Para ver por que, observe que, como a árvore de instâncias de procedimento recursivo é uma árvore binária cheia, o número de nós internos é uma unidade menor que o número de folhas (veja o Exercício B.5-3). Cada nó interno realiza trabalho constante para dividir a faixa de iteração, e cada folha corresponde a uma iteração do laço, que demora no mínimo o tempo constante ((n) nesse caso). Assim, podemos amortizar a sobrecarga de geração recursiva em relação ao trabalho das iterações, contribuindo no máximo com um fator constante para o trabalho global.

Figura 27.4 Um gad que representa a computação de MAT-VEC-MAIN-LOOP(A, x, y, 8, 1, 8). Os dois números dentro de cada retângulo com cantos arredondados dão os valores dos dois últimos parâmetros (i e i' no cabeçalho do procedimento) na invocação (geração ou chamada) do procedimento. Os círculos pretos representam fibras que correspondem ou ao caso-base ou à parte do procedimento até a geração de MAT-VEC-MAIN-LOOP na linha 5; os círculos sombreados representam fibras que correspondem à parte do procedimento que chama MAT-VEC--MAIN-LOOP na linha 6 até a **sync** na linha 7, onde é suspenso até que a sub-rotina gerada na linha 5 retorne; e os círculos brancos representam fibras que correspondem à parte (desprezível) do procedimento após a **sync** até o ponto onde ela retorna.

Como questão prática, aplicar multithread dinâmico a plataformas de concorrência, às vezes, **adensa** as folhas da recursão por executar várias iterações em uma única folha, seja automaticamente, seja sob controle do programador, reduzindo assim a sobrecarga da geração recursiva. Todavia, essa sobrecarga reduzida é conseguida à custa de reduzir também o paralelismo. Porém, se a computação tem folga paralela suficiente, o fator de aceleração linear quase perfeito não precisa ser sacrificado.

Também temos de levar em conta a sobrecarga de geração recursiva quando analisamos a duração de um constructo de laço paralelo. Visto que a profundidade da chamada recursiva é logarítmica em relação ao número de iterações, para um laço paralelo com n iterações no qual a i-ésima iteração tem duração $iter_\infty(i)$, a duração é

$$T_\infty(n) = \Theta(\lg n) = \max_{1 \le i \le n} iter_\infty(i).$$

Por exemplo, para MAT-VEC em uma matriz $n \times n$, o laço de inicialização paralelo nas linhas 3–4 tem duração (lg n) porque a geração recursiva domina o trabalho de tempo constante de cada iteração. A duração dos laços duplamente aninhados nas linhas 5–7 é (n) porque cada iteração do laço **parallel for** externo contém n iterações do laço **for** interno (serial). A duração do código restante no procedimento é constante e, assim, a duração é dominada pelos laços duplamente aninhados, o que produz uma duração global de (n) para todo o procedimento. Visto que o trabalho é (n_2), o paralelismo é (n_2)/(n) = (n). (O Exercício 27.1-6 pede que você dê uma implementação que tenha paralelismo ainda maior.)

Condições de corrida

Um algoritmo multithread é ***determinístico*** se sempre faz a mesma coisa na mesma entrada, não importando como as instruções são escalonadas no computador multinúcleo. É ***não determinístico*** se seu comportamento pode variar de execução para execução. Muitas vezes, um algoritmo multithread que deveria ser determinístico deixa de ser porque contém uma "corrida de determinância".

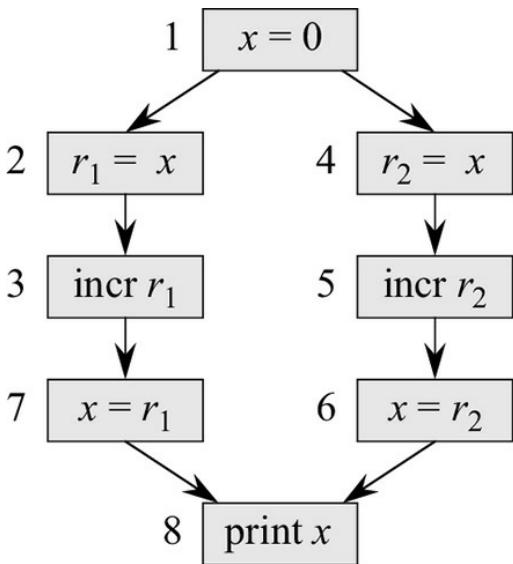
Condições de corrida são os venenos da concorrência. Entre os bugs famosos de corrida citamos o aparelho de radioterapia Therac-25, que matou três pessoas e feriu várias outras, e o blecaute que ocorreu nos Estados Unidos em 2003, que deixou mais de 50 milhões de pessoas sem energia elétrica. Esses bugs perniciosos são notoriamente difíceis

de descobrir. Você pode executar testes em laboratório dias a fio sem ocorrer nenhuma falha, só para descobrir que seu software quebra esporadicamente em campo.

Uma ***corrida de determinância*** ocorre quando duas instruções logicamente paralelas acessam a mesma localização de memória, e no mínimo uma das instruções faz uma gravação. O seguinte procedimento ilustra a condição de corrida:

RACE-EXAMPLE( )
1 $x = 0$
2 **parallel for** $i = 1$ **to** 2
3 $x = x + 1$
4 print x

Após inicializar x como 0 na linha 1, RACE-EXAMPLE cria duas fibras paralelas e cada uma incrementa x na linha 3. Embora aparentemente RACE-EXAMPLE deva sempre imprimir o valor 2 (sua serialização certamente o faz), poderia imprimir o valor 1. Vamos ver como essa anomalia poderia ocorrer.

Quando um processador incrementa x, a operação não é indivisível, mas é composta por uma sequência de instruções:

1. Leia x da memória para um dos registradores do processador.
2. Incremente o valor no registrador.
3. Grave o valor no registrador de volta em x na memória.

A Figura 27.5(a) ilustra um gad de computação que representa a execução de RACE-EXAMPLE, com as fibras decompostas em instruções individuais. Lembre-se de que, como um computador paralelo ideal suporta consistência sequencial, podemos ver a execução paralela de um algoritmo multithread como uma intercalação de instruções que respeita as dependências no gad. A parte (b) da figura mostra os valores em uma execução da computação que revela a anomalia. O valor x é armazenado na memória, e r_1 e r_2 são registradores de processador. Na etapa 1, um dos processadores atribui 0 a x. Nas etapas 2 e 3, o processador 1 lê x da memória para seu registrador r_1 e o incrementa, produzindo o valor 1 em r_1. Nesse ponto, o processador 2 entra em cena e executa as instruções 4–6. O processador 2 lê x da memória para o registrador r_2; incrementa o registrador, produzindo o valor 1 em r_2; depois armazena esse valor em x, definindo x como 1. Agora, o processador 1 retoma a etapa 7, armazenando o valor 1 em r_1 em x, o que deixa o valor de x inalterado. Portanto, a etapa 8 imprime o valor 1, em vez de 2, como a serialização imprimiria.

Podemos ver o que aconteceu. Se o efeito da execução paralela fosse que o processador 1 executa todas as suas instruções antes do processador 2, o valor 2 seria impresso.

Ao contrário, se o efeito fosse que o processador 2 executa todas as suas instruções antes do processador 1, o valor 2 ainda seria impresso. Contudo, quando as instruções dos dois processadores são executadas ao mesmo tempo, é possível, como nesse exemplo de execução, que uma das atualizações para x seja perdida.

É claro que muitas das execuções não expõem o bug. Por exemplo, se a ordem da execução fosse $\langle 1, 2, 3, 7, 4, 5, 6, 8 \rangle$ ou $\langle 1, 4, 5, 6, 2, 3, 7, 8 \rangle$, obteríamos o resultado correto. Esse é o problema das corridas de determinância. Em geral, a maioria das ordenações produz resultados corretos — como qualquer uma na qual as instruções à esquerda são executadas antes das instruções à direita ou vice-versa. Porém, algumas ordenações geram resultados impróprios quando as instruções se intercalam. Consequentemente, pode ser muito difícil testar corridas. Você pode executar testes dias a fio e nunca ver o bug, só para sofrer uma catastrófica queda do sistema em campo quando o resultado é crítico.

etapa	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

Figura 27.5 Ilustração da corrida de determinância em RACE-EXAMPLE. **(a)** Um gad de computação que mostra as dependências entre instruções individuais. Os registradores do processador são r_1 e r_2. Instruções não relacionadas com a corrida, como a implementação de controle de laço, são omitidas. **(b)** Uma sequência de execução que revela o bug, mostrando os valores de x na memória e nos registradores r_1 e r_2 para cada etapa na sequência de execução.

Embora haja vários modos possíveis de enfrentar corridas, incluindo utilizar travas de exclusão mútua e outros métodos de sincronização, para a nossa finalidade simplesmente garantiremos que filamentos que funcionam em paralelo são ***independentes***: não existe nenhuma corrida de determinância entre eles. Assim, em um constructo **parallel for**, todas as iterações devem ser independentes. Entre uma **spawn** e a **sync** correspondente, o código do filho gerado deve ser independente do código do pai, incluindo código executado por filho adicional gerado ou chamado. Observe que argumentos para um filho gerado são avaliados no pai antes de ocorrer a geração propriamente dita e, assim, a avaliação de argumentos para uma sub-rotina gerada é em série com quaisquer acessos àqueles argumentos após a geração.

Como exemplo que mostra como é fácil gerar código com corridas, apresentamos uma implementação defeituosa de multiplicação multithread de matriz por vetor que alcança uma duração de (lg n) pela paralelização do laço **for** interno:

MAT-VEC-WRONG(A, x)
1 $n = A.linhas$
2 seja y um novo vetor de comprimento n
3 **parallel for** $i = 1$ **to** n
4 $y_i = 0$
5 **parallel for** $i = 1$ **to** n
6 **parallel for** $j = 1$ **to** n
7 $y_i = y_i + a_{ij} x_j$
8 **return** y

Infelizmente, esse procedimento é incorreto por causa de corridas na atualização y_i na linha 7, que é executada concorrentemente para todos os n valores de j. O Exercício 27.1-6 pede que você dê uma implementação correta com duração (lg n).

Às vezes, um algoritmo multithread com corridas pode estar correto. Como exemplo, dois threads paralelos poderiam armazenar o mesmo valor em uma variável compartilhada e não importaria qual armazenaria o valor antes. Todavia, em geral, consideraremos códigos com corridas ilegais.

Uma aula de xadrez

Concluímos esta seção com uma história verdadeira que ocorreu durante o desenvolvimento do programa multithread de classe mundial para jogar xadrez denominado ★Socrates [81], embora os tempos a seguir tenham sido simplificados para essa demonstração. O programa foi prototipado em um computador com 32 processadores, mas afinal era para ser executado em um supercomputador com 512 processadores. A certa altura, os desenvolvedores incorporaram uma otimização no programa que reduzia seu tempo de execução em relação a um importante padrão de comparação estabelecido na máquina de 32 processadores de T_32 = 65 segundos a T'_32 = 40 segundos. No entanto, os desenvolvedores usaram as medidas de desempenho de trabalho e duração para concluir que a versão otimizada, que era mais rápida com 32 processadores, na verdade seria mais lenta que a versão original com 512 processadores. O resultado é que eles abandonaram essa "otimização".

Apresentamos a seguir a análise que eles fizeram. A versão original do programa tinha trabalho T_1 = 2048 segundos e duração T_∞ = 1 segundo. Se tratarmos a desigualdade (27.4) como uma equação, $T_P = T_1/P + T_\infty$, e a usarmos como uma aproximação para o tempo de execução com P processadores, veremos que, de fato, T_{32} = 2048/32 + 1 = 65. Com a otimização, o trabalho tornou-se T'_1 = 1024 segundos e a duração tornou-se T'_∞ = 8 segundos. Utilizando novamente nossa aproximação, obtemos T'_{32} = 1024/32 + 8 = 40.

Todavia, as velocidades relativas das duas versões mudam quando calculamos os tempos de execução com 512 processadores. Em particular, temos T_{512} = 2048/512 + 1 = 5 segundos e T'_{512} = 1024/512 + 8 = 10 segundos. Então, a otimização que acelerou o programa com 32 processadores faria com que o programa ficasse duas vezes mais lento com 512 processadores! A duração 8 da versão otimizada, que não era o termo dominante no tempo de execução com 32 processadores, tornou-se o termo dominante com 512 processadores, anulando a vantagem de utilizar mais processadores.

A moral da história é que trabalho e duração podem nos dar um modo melhor de extrapolar desempenho que tempos de execução medidos.

Exercícios

27.1-1 Suponha que geramos P-FIB(n – 2) na linha 4 de P-FIB, em vez de chamá-la como é feito no código. Qual é o impacto no trabalho, na duração e no paralelismo assintóticos?

27.1-2 Desenhe o gad de computação que resulta de executar P-FIB(5). Considerando que cada fibra na computação demora tempo unitário, quais são o trabalho, a duração e o paralelismo da computação? Mostre como escalonar o gad em três processadores utilizando escalonamento guloso rotulando cada filamento com a etapa de tempo na qual é executado.

27.1-3 Prove que um escalonador guloso consegue o seguinte limite de tempo, que é ligeiramente mais forte que o limite provado no Teorema 27.1:

$$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty. \tag{27.5}$$

27.1-4 Construa um gad de computação para o qual uma execução de um escalonador guloso pode demorar quase duas vezes o tempo de uma outra execução de um escalonador guloso com o mesmo número de processadores. Descreva como ocorreriam as duas execuções.

27.1-5 A professora Karan mede seu algoritmo multithread determinístico com 4, 10 e 64 processadores de um computador paralelo ideal utilizando um escalonador guloso. Diz ela que as três execuções produziram T_4 = 80 segundos, T_{10} = 42 segundos e T_{64} = 10 segundos. Demonstre que a professora está mentindo ou é incompetente. (*Sugestão:* Use a lei do trabalho (27.2), a lei da duração (27.3) e a desigualdade (27.5) do Exercício 27.1-3.)

27.1-6 Dê um algoritmo multithread para multiplicar uma matriz $n \times n$ por um n-vetor que consegue paralelismo $(n_2/\lg n)$, mantendo, ao mesmo tempo, trabalho (n_2).

27.1-7 Considere o seguinte pseudocódigo multithread para transpor uma matriz $n \times n$ A no lugar:

```
P-TRANSPOSE(A)
1 n = A.linhas
2 parallel for j = 2 to n
3         parallel for i = 1 to j − 1
4                 troque a_ij por a_ji
```

Analise o trabalho, a duração e o paralelismo desse algoritmo.

27.1-8 Suponha que substituímos o laço **parallel for** na linha 3 de P-TRANSPOSE (veja o Exercício 27.1-7) por um laço **for** comum. Analise o trabalho, a duração e o paralelismo do algoritmo resultante.

27.1-9 Para quantos processadores as duas versões dos programas de jogo de xadrez são executadas com a mesma rapidez, considerando que $T_P = T_1/P + T_\infty$?

27.2 MULTIPLICAÇÃO MULTITHREAD DE MATRIZES

Nesta seção, examinamos como aplicar multithread a uma multiplicação de matrizes, um problema cujo tempo de execução serial estudamos na Seção 4.2. Examinaremos algoritmos multithread baseados no laço-padrão triplamente aninhado, bem como em algoritmos de divisão e conquista.

Multiplicação multithread de matrizes

O primeiro algoritmo que estudamos é o algoritmo direto baseado na paralelização dos laços no procedimento SQUARE-MATRIX-MULTIPLY na página 75:

```
P-SQUARE-MATRIX-MULTIPLY(A,B)
1      n = A.rows
2      seja C uma nova matriz n × n
3      parallel for i = 1 to n
4              parallel for j = 1 to n
5                      c_ij = 0
6                      for k = 1 to n
7                              c_ij = c_ij + a_ik · b_kj
8      return C
```

Para analisar esse algoritmo, observe que, como a serialização do algoritmo é exatamente SQUARE-MATRIX-MULTIPLY, o trabalho é, portanto, simplesmente $T_1(n) = (n_3)$, igual ao tempo de execução de SQUARE-MATRIX-MULTIPLY. A duração é $T\infty(n) = (n)$, porque segue um caminho que desce pela árvore de recursão para o laço **parallel for** que começa na linha 3, depois desce pela árvore de recursão para o laço **parallel for** que começa na linha 4 e executa todas as n iterações do laço **for** comum, que começa na linha 6, resultando em duração total de $(\lg n) + (\lg n) + (n) = (n)$. Assim, o paralelismo é $(n_3)/(n) = (n_2)$. O Exercício 27.2-3 pede que se paralelize o laço interno para obter um paralelismo de $(n_3/\lg n)$, o que não se pode fazer diretamente utilizando **parallel for** porque criaria corridas.

Algoritmo multithread de divisão e conquista para multiplicação de matrizes

Como aprendemos na Seção 4.2, podemos multiplicar matrizes $n \times n$ serialmente no tempo $(n_{\lg}\ 7) = O(n_2,{}^{81})$ utilizando a estratégia de divisão e conquista de Strassen, que nos motiva a considerar a aplicação de multithread a tal algoritmo. Começamos, como fizemos na Seção 4.2, com a aplicação de multithread a um algoritmo de divisão e conquista mais simples.

Lembre-se de que vimos, na Seção 4.2, que o procedimento SQUARE-MATRIX-MULTIPLY-RECURSIVE, que multiplica duas matrizes $n \times n$ A e B para produzir a matriz $n \times n$ C, recorre ao particionamento de cada uma das três matrizes em quatro submatrizes $n/2 \times n/2$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Então, podemos escrever o produto de matrizes como

$$\begin{aligned} \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}. \end{aligned} \tag{27.6}$$

Assim, para multiplicar duas matrizes $n \times n$, efetuamos oito multiplicações de matrizes

$n/2 \times n/2$ e uma adição de matrizes $n \times n$. O pseudocódigo apresentado a seguir implementa essa estratégia de divisão e conquista utilizando paralelismo aninhado. Diferentemente do procedimento SQUARE-MATRIX-MULTIPLY-RECURSIVE no qual é baseado, P-MATRIX-MULTIPLY-RECURSI- VE adota a matriz de saída como parâmetro para evitar alocar matrizes desnecessariamente.

```
P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)
1    n = A.linhas
2    if n == 1
3        c11 = a11 b11
4    else seja T uma nova matriz n × n
5        particione A, B, C e T em submatrizes n/2 × n/2
             A11,A12,A21,A22, B11,B12,B21,B22, C11,C12,C21,C22,
             e T11, T12, T21, T22, respectivamente
6        spawn P-MATRIX-MULTIPLY-RECURSIVE(C11,A11,B11)
7        spawn P-MATRIX-MULTIPLY-RECURSIVE(C12,A11,B12)
8        spawn P-MATRIX-MULTIPLY-RECURSIVE(C21,A21,B11)
9        spawn P-MATRIX-MULTIPLY-RECURSIVE(C22,A21,B12)
10       spawn P-MATRIX-MULTIPLY-RECURSIVE(T11,A12,B21)
11       spawn P-MATRIX-MULTIPLY-RECURSIVE(T12,A12,B22)
12       spawn P-MATRIX-MULTIPLY-RECURSIVE(T21,A22,B21)
13       P-MATRIX-MULTIPLY-RECURSIVE(T22,A22,B22)
14       sync
15       parallel for i = 1 to n
16           parallel for j = 1 to n
17               cij = cij + tij
```

A linha 3 trata do caso-base, onde estamos multiplicando matrizes 1×1. Tratamos o caso recursivo nas linhas 4–17. Alocamos uma matriz temporária T na linha 4, e a linha 5 particiona cada uma das matrizes A, B, C e T em submatrizes $n/2 \times n/2$. (Como ocorre com SQUARE-MATRIX-MULTIPLY-RECURSIVE na Seção 4.2, atenuamos a questão de menor importância, que é como usar cálculos de índices para representar seções de submatriz de uma matriz.) A chamada recursiva na linha 6 define a submatriz C_{11} como o produto de submatrizes $A_{11}B_{11}$, de modo que C_{11} é igual ao primeiro dos dois termos que formam sua soma na equação (27.6). De modo semelhante, as linhas 7–9 definem C_{12}, C_{21} e C_{22} como o primeiro dos dois termos que são iguais às suas somas na equação (27.6). A linha 10 define a submatriz T_{11} como o produto de submatrizes

$A_{12}B_{21}$, de modo que T_{11} é igual ao segundo dos dois termos que formam a soma de C_{11}. As linhas 11–13 definem T_{12}, T_{21} e T_{22} como o segundo dos dois termos que formam a soma de C_{12}, C_{21} e C_{22}, respectivamente. As primeiras sete chamadas recursivas são geradas, e a última é executada na fibra principal. A declaração **sync** na linha 14 garante que todos os subprodutos de matrizes nas linhas 6–13 foram calculados, depois disso, acrescentamos os produtos de T em C utilizando os laços **parallel for** duplamente aninhados nas linhas 15–17.

Primeiro analisamos o trabalho $M_1(n)$ do procedimento P-MATRIX-MULTIPLY-RECURSIVE, ecoando a análise do tempo de execução serial de seu progenitor SQUARE-MATRIX-MULTIPLY-RECURSIVE. No caso recursivo, particionamos no tempo (1), executamos oito multiplicações recursivas de matrizes $n/2 \times n/2$, e encerramos com o trabalho (n_2) resultante da soma de duas matrizes $n \times n$. Assim, a recorrência para o trabalho $M_1(n)$ é

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3) \end{aligned}$$

pelo caso 1 do teorema mestre. Em outras palavras, o trabalho de nosso algoritmo multithread é assintoticamente o mesmo que o tempo de execução do procedimento SQUARE-MATRIX-MULTIPLY na Seção 4.2, com seus laços triplamente aninhados.

Para determinar a duração $M\infty(n)$ de P-MATRIX-MULTIPLY-RECURSIVE, primeiro observamos que a duração para particionamento é (1), que é dominado pela duração (lg n) do laço **parallel for** duplamente aninhado nas linhas 15–17. Como todas as oito chamadas recursivas paralelas são executadas em matrizes do mesmo tamanho, a duração máxima para qualquer chamada recursiva é exatamente a duração de qualquer uma. Por consequência, a recorrência para a duração $M_1(n)$ de P-MATRIX-MULTIPLY-RECURSIVE é

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n). \quad (27.7)$$

Essa recorrência não se enquadra em nenhum dos casos do teorema mestre, mas obedece à condição do Exercício 4.6-2. Portanto, pelo Exercício 4.6-2, a solução para a recorrência (27.7) é $M_\infty(n) = (\lg^2 n)$.

Agora, que conhecemos o trabalho e a duração de P-MATRIX-MULTIPLY-RECURSIVE, podemos calcular seu paralelismo como $M_1(n)/M_\infty(n) = (n^3 / \lg^2 n)$, que é muito alto.

Aplicar multithread ao método de Strassen

Para aplicar multithread ao algoritmo de Strassen, seguimos as mesmas linhas gerais da página 79, só que utilizando paralelismo aninhado:

1. Divida as matrizes de entrada A e B e a matriz de saída C em submatrizes $n/2 \times n/2$, como na equação (27.6). Essa etapa demora trabalho (1) e duração por cálculo de índice.
2. Crie 10 matrizes $S_1, S_2, \ldots, S_{10}$, cada uma delas sendo $n/2 \times n/2$ e a soma ou diferença das duas matrizes criadas na etapa 1. Podemos criar todas as 10 matrizes com trabalho (n_2) e duração $(\lg n)$ utilizando laços **parallel for** duplamente aninhados.
3. Utilizando as submatrizes criadas na etapa 1 e as 10 matrizes criadas na etapa 2, gere recursivamente a computação dos sete produtos de matrizes $n/2 \times n/2$ $P_1, P_2, \ldots, P_7$.
4. Calcule as submatrizes desejadas C_{11}, C_{12}, C_{21}, C_{22} da matriz de resultados C somando e subtraindo várias combinações das P_i matrizes utilizando, mais uma vez, laços **parallel for** duplamente aninhados. Podemos calcular todas as quatro submatrizes com trabalho (n_2) e duração $(\lg n)$.

Para analisar esse algoritmo, primeiro observamos que a serialização é igual à do algoritmo serial original, e o trabalho é exatamente o tempo de execução da serialização, isto é, $(n_{\lg} 7)$. Quanto a P-MATRIX-MULTIPLY-RECURSIVE, podemos criar uma recorrência para a duração. Nesse caso, sete chamadas recursivas são executadas em paralelo, porém, visto que todas elas funcionam em matrizes do mesmo tamanho, obtemos a mesma recorrência (27.7) que obtivemos para P-MATRIX-MULTIPLY-RECURSIVE, que tem solução $(\lg^2 n)$. Assim, o paralelismo do método >multithread de Strassen é $(n_{\lg} 7/ \lg^2 n)$, que é alto, embora ligeiramente menor que o paralelismo de P-MATRIX-MULTIPLY-RECURSIVE.

Exercícios

27.2-1 Desenhe o gad de computação para calcular P-SQUARE-MATRIX-MULTIPLY de matrizes 2×2, identificando como os vértices em seu diagrama correspondem a fibras na execução do algoritmo. Use a seguinte convenção: arestas geradas e arestas de chamada apontam para baixo, arestas de continuação apontam para a direita na horizontal e arestas de retorno apontam para cima. Supondo que cada fibra demora tempo unitário, analise o trabalho, a duração e o paralelismo dessa computação.

27.2-2 Repita o Exercício 27.2-1 para P-MATRIX-MULTIPLY-RECURSIVE.

27.2-3 Dê pseudocódigo para um algoritmo multithread que multiplica duas matrizes $n \times n$ com trabalho (n_3) mas duração de somente $(\lg n)$. Analise seu algoritmo.

27.2-4 Dê pseudocódigo para um algoritmo multithread eficiente que multiplica uma matriz $p \times q$ por uma matriz $q \times r$. Seu algoritmo deve ter alto grau de paralelismo, mesmo se qualquer das p, q e r for 1. Analise seu algoritmo.

27.2-5 Dê pseudocódigo para um algoritmo multithread eficiente que transponha uma matriz $n \times n$ no local utilizando divisão e conquista para dividir a matriz recursivamente em quatro submatrizes $n/2 \times n/2$. Analise seu algoritmo.

27.2-6 Dê pseudocódigo para uma implementação multithread eficiente do algoritmo de Floyd-Warshall (veja a Seção 25.2), que calcula caminhos mínimos entre todos os pares de vértices em um grafo de arestas ponderadas. Analise seu algoritmo.

27.3 ORDENAÇÃO POR INTERCALAÇÃO MULTITHREAD

Primeiro vimos ordenação por intercalação serial na Seção 2.3.1, e na Seção 2.3.2 analisamos seu tempo de execução e mostramos que é ($n \lg n$). Como a ordenação por intercalação já usa o paradigma de divisão e conquista, parece ser uma candidata espetacular para multithread por paralelismo aninhado. É fácil modificar o pseudocódigo para gerar a primeira chamada recursiva:

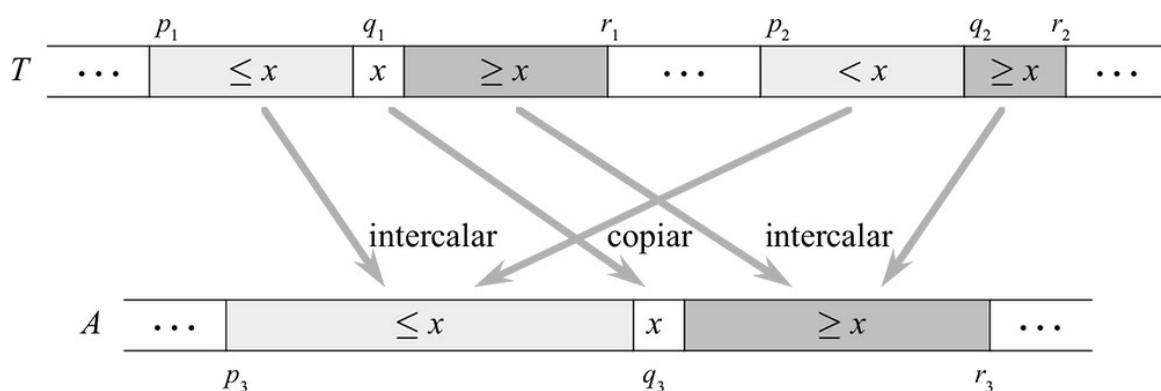
```
MERGE-SORT'(A, p, r)
1   if p < r
2       q = ⌊(p + r)/2⌋
3       spawn MERGE-SORT'(A, p, q)
4       MERGE-SORT'(A, q + 1, r)
5       sync
6       MERGE(A, p, q, r)
```

Como sua contraparte serial, MERGE-SORT' ordena o subarranjo $A[p \,..\, r]$. Após a conclusão das duas sub-rotinas recursivas nas linhas 3 e 4, garantida pela declaração **sync** na linha 5, MERGE-SORT' chama o mesmo procedimento MERGE chamado na página 31.

Vamos analisar MERGE-SORT'. Para tal, primeiro precisamos analisar MERGE. Lembre-se de que seu tempo de execução serial para intercalar n elementos é (n). Como MERGE é serial, seu trabalho, bem como sua duração, são (n). Assim, a seguinte recorrência caracteriza o trabalho $MS'_1(n)$ de MERGE-SORT' para n elementos:

$$\begin{aligned} MS'_1(n) &= 2MS'_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n), \end{aligned}$$

que é igual ao tempo de execução serial de ordenação por intercalação. Visto que as duas chamadas recursivas de MERGE-SORT' podem ser executadas em paralelo, a duração MS'_∞ é dada pela recorrência

$$\begin{aligned} MS'_\infty(n) &= MS'_\infty(n/2) + \Theta(n) \\ &= \Theta(n). \end{aligned}$$

Assim, o paralelismo de MERGE-SORT' chega a $MS'_1(n)/MS'_\infty(n)$ = ($\lg n$), que não é um grau impressionante de paralelismo. Para ordenar 10 milhões de elementos, por exemplo, ele poderia alcançar fator de aceleração linear com um pequeno número de processadores, mas a escala não aumentaria efetivamente com centenas de processadores.

É provável que você já tenha uma ideia de onde se encontra o gargalo do paralelismo nessa ordenação por intercalação multithread: o procedimento serial MERGE. Embora de início a intercalação pareça ser inerentemente serial, na verdade podemos criar uma versão multithread dela utilizando paralelismo aninhado.

Nossa estratégia de divisão e conquista para intercalação multithread, que é ilustrada na Figura 27.6, funciona em subarranjos de um arranjo T. Suponha que estejamos intercalando os dois subarranjos ordenados $T[p_1 \,..\, r_1]$ de comprimento $n_1 = r_1 - p_1 + 1$ e $T[p_2 \,..\, r_2]$ de comprimento $n_2 = r_2 - p_2 + 1$ em um outro subarranjo $A[p_3 \,..\, r_3]$, de comprimento $n_3 = r_3 - p_3 + 1 = n_1 + n_2$. Sem perda de generalidade, adotamos a hipótese simplificadora $n_1 \geq n_2$.

Em primeiro lugar determinamos o elemento do meio $x = T[q_1]$ do subarranjo $T[p_1 \,..\, r_1]$, onde $q_1 = (p_1 + r_1)/2$. Como o subarranjo está ordenado, x é uma mediana de $T[p_1 \,..\, r_1]$: nenhum elemento em $T[p_1 \,..\, q_1 - 1]$ é maior que

x e nenhum elemento em $T[q_1 + 1 \,..\, r_1]$ é menor que x. Então, usamos busca binária para encontrar o índice q_2 no subarranjo $T[p_2 \,..\, r_2]$ de modo que o subarranjo ainda estaria ordenado se inseríssemos x entre $T[q_2 - 1]$ e $T[q_2]$.

Figura 27.6 A ideia que fundamenta a intercalação multithread de dois subarranjos ordenados $T p_1 \,..\, r_1$ e $T p_2 \,..\, r_2$ para o subarranjo $A p_3 \,..\, r_3$. Fazendo $x = T q_1$ a mediana de $T p_1 \,..\, r_1$ e q_2 o lugar em $T p_2 \,..\, r_2$ tal que x cairia entre $T q_2$ - 1 e $T q_2$, todo elemento nos subarranjos $T p_1 \,..\, q_1$ - 1 e $T p_2 \,..\, q_2$ - 1 (sombreado em tom mais claro) é menor ou igual a x, e todo elemento nos subarranjos $T q_1 + 1 \,..\, r_1$ e $T q_2 + 1 \,..\, r_2$ (sombreado em tom mais escuro) é no mínimo x. Para intercalar, calculamos o índice q_3 do local em $A p_3 \,..\, r_3$ ao qual x pertence, copiamos x para $A q_3$, e então intercalamos recursivamente $T p_1 \,..\, q_1$ - 1 com $T p_2 \,..\, q_2$ - 1 em $A p_3 \,..\, q_3$ - 1 e $T q_1 + 1 \,..\, r_1$ com $T q_2 \,..\, r_2$ em $A q_3 + 1 \,..\, r_3$.

Em seguida, intercalamos os subarranjos originais $T[p_1 \,..\, r_1]$ e $T[p_2 \,..\, r_2]$ em $A[p_3 \,..\, r_3]$ da seguinte maneira:

1. Defina $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$.
2. Copie x para $A q_3$.
3. Intercale recursivamente $T p_1 \,..\, q_1$-1 com $T p_2 \,..\, q_2$-1 e coloque o resultado no subarranjo $A p_3 \,..\, q_3$ - 1.
4. Ordene recursivamente $T q_1 + 1 \,..\, r_1$ com $T q_2 \,..\, r_2$ e coloque o resultado no subarranjo $A[q_3 + 1 \,..\, r_3]$.

Quando calculamos q_3, a quantidade $q_1 – p_1$ é o número de elementos no subarranjo $T[p_1 \,..\, q_1 - 1]$, e a quantidade $q_2 - p_2$ é o número de elementos no subarranjo $T[p_2 \,..\, q_2 - 1]$. Assim, a soma das duas é o número de elementos que acabam ficando antes de x no subarranjo $A[p_3 \,..\, r_3]$.

O caso-base ocorre quando $n_1 = n_2 = 0$ e não temos nenhum trabalho a fazer para intercalar os dois subarranjos vazios. Visto que supomos que o subarranjo $T[p_1 \,..\, r_1]$ é no mínimo tão comprido quanto $T[p_2 \,..\, r_2]$, isto é, $n_1 \geq n_2$, para verificar se é o caso-base basta verificar se $n_1 = 0$. Devemos também garantir que a recursão trate adequadamente do caso em que um dos dois subarranjos é vazio, o qual, como supomos que $n_1 \geq n_2$, deve ser o subarranjo $T[p_2 \,..\, r_2]$.

Agora, vamos passar essas ideias para pseudocódigo. Começamos com a busca binária, que expressamos serialmente. O procedimento BINARY-SEARCH(x, T, p, r) toma uma chave x e um subarranjo $T[p \,..\, r]$ e retorna um dos seguintes:

- Se $T p \,..\, r$ é vazio ($r < p$), então ele devolve o índice p.
- Se $x \leq T p$ e, por consequência, menor ou igual a todos os elementos de $T p \,..\, r$, então ele devolve o índice p.
- Se $x > T p$, então ele devolve o maior índice q na faixa $p < q \leq r + 1$ tal que $T q$ - $1 < x$.

Este é o pseudocódigo:

```
BINARY-SEARCH(x, T, p, r)
1   low = p
2   high = max(p, r + 1)
3   while low < high
4       mid = ⌊(low + high)/2⌋
5       if x ≤ T [mid]
6           high = mid
7       else low = mid + 1
8   return high
```

A chamada BINARY-SEARCH(x, T, p, r) demora tempo serial (lg n) no pior caso, onde $n = r - p = 1$ é o tamanho do subarranjo no qual ela é executada (veja o Exercício 2.3-5). Visto que BINARY-SEARCH é um procedimento serial, o trabalho e a duração de seu pior caso são (lg n).

Agora estamos preparados para escrever pseudocódigo para o procedimento de intercalação multithread em si. Como o procedimento MERGE na página 31, o procedimento P-MERGE supõe que os dois subarranjos que devem ser intercalados encontram-se no mesmo arranjo. Entretanto, diferentemente de MERGE, P-MERGE não supõe que os dois subarranjos que devem ser intercalados são adjacentes dentro do arranjo (isto é, P-MERGE não requer que $p_2 = r_1 + 1$). Uma outra diferença entre MERGE e P-MERGE é que P-MERGE adota como argumento um subarranjo de saída A no qual os valores intercalados devem ser armazenados. Uma chamada P-MERGE (T, p_1, r_1, p_2, r_2, A, p_3) intercala os subarranjos ordenados $T[p_1 \,..\, r_1]$ e $T[p_2 \,..\, r_2]$ no subarranjo $A[p_3 \,..\, r_3]$, onde $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$ e não é dado como uma entrada.

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
1   n1 = r1 − p1 + 1
2   n2 = r2 − p2 + 1
3   if n1 < n2                          // assegure que n1 ≥ n2
4       troque p1 com p2
5       troque r1 com r2
6       troque n1 com n2
7   if n1 == 0                          //ambos vazios?
8       return
9   else q1 = ⌊(p1 + r1)/2⌋
10      q2 = BINARY-SEARCH(T [q1], T, p2, r2)
11      q3 = p3 + (q1 − p1) + (q2 − p2)
12      A[q3] = T [q1]
13      spawn P-MERGE(T, p1, q1 − 1, p2, q2 − 1, A, p3)
14      P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
15      sync
```

O procedimento P-MERGE funciona da seguinte maneira: as linhas 1–2 calculam os comprimentos n_1 e n_2 dos subarranjos $T[p_1 \,..\, r_1]$ e $T[p_2 \,..\, r_2]$, respectivamente. As linhas 3–6 forçam a condição $n_1 \geq n_2$. A linha 7 testa se é o caso-base, onde o subarranjo $T[p_1 \,..\, r_1]$ é vazio (e, por consequência, também $T[p_2 \,..\, r_2]$ é vazio), caso em que simplesmente retornamos. As linhas 9–15 implementam a estratégia de divisão e conquista. A linha 9 calcula o ponto do meio de $T[p_1 \,..\, r_1]$ e a linha 10 encontra o ponto q_2 em $T[p_2 \,..\, r_2]$ tal que todos os elementos em $T[p_2 \,..\, q_2 - 1]$ são menores que $T[q_1]$ (que corresponde a x) e todos os elementos em $T[q_2 \,..\, r_2]$ são no mínimo tão grandes quanto $T[q_1]$. A linha 11 calcula o índice q_3 do elemento que divide o subarranjo de saída $A[p_3 \,..\, r_3]$ em $A[p_3 \,..\, q_3 - 1]$ e $A[q_3 + 1 \,..\, r_3]$, e então a linha 12 copia $T[q_1]$ diretamente para $A[q_3]$.

Então, executamos recursão utilizando paralelismo aninhado. A linha 13 gera o primeiro subproblema, enquanto a linha 14 chama o segundo subproblema em paralelo. A declaração **sync** na linha 15 garante que os subproblemas estão concluídos antes de o procedimento retornar. (Visto que todo procedimento implicitamente executa **sync** antes de retornar, poderíamos ter omitido a declaração **sync** na linha 15, mas incluí-la é boa prática de codificação.) Há certa esperteza na codificação para garantir que, quando o subarranjo $T[p_2 \,..\, r_2]$ é vazio, o código funciona corretamente. O modo como funciona é que, em cada chamada recursiva, um elemento mediana de $T[p_1 \,..\, r_1]$ é colocado no subarranjo de saída até que o próprio $T[p_1 \,..\, r_1]$ finamente se torna vazio, o que aciona o caso-base.

Análise de intercalação multithread

Primeiro deduzimos uma recorrência para a duração $PM_\infty(n)$ de P-MERGE, onde os dois subarranjos contêm um total de $n = n_1+n_2$ elementos. Como a geração na linha 13 e a chamada na linha 14 funcionam logicamente em paralelo, basta examinar somente a chamada que custa mais. A chave é entender que, no pior caso, o número de elementos em qualquer das chamadas recursivas pode ser no máximo $3n/4$, o que verificamos da seguinte maneira: como as linhas 3–6 garantem que $n_2 \le n_1$, decorre que $n_2 = 2n_2/2 \le (n_1 + n_2)/2 = n/2$. No pior caso, uma das duas chamadas recursivas intercala $n_1/2$ elementos de $T[p_1 \,..\, r_1]$ com todos os n_2 elementos de $T[p_2 \,..\, r_2]$ e, por consequência, o número de elementos envolvidos na chamada é

$$\begin{aligned} \lfloor n_1/2 \rfloor + n_2 &\le n_1/2 + n_2/2 + n_2/2 \\ &= (n_1 + n_2)/2 + n_2/2 \\ &\le n/2 + n/4 \\ &= 3n/4. \end{aligned}$$

Somando também o custo (lg n) da chamada a BINARY-SEARCH na linha 10, obtemos a seguinte recorrência para a duração do pior caso:

$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\lg n). \tag{27.8}$$

(Para o caso-base, a duração é (1), visto que as linhas 1–8 são executadas em tempo constante.) Essa recorrência não se enquadra em nenhum dos casos do teorema mestre, mas cumpre a condição do Exercício 4.6-2. Portanto, a solução para a recorrência (27.8) é $PM_1(n) = (\lg^2 n)$.

Agora, analisamos o trabalho $PM_1(n)$ de P-MERGE para n elementos, que é (n). Visto que cada um dos n elementos deve ser copiado do arranjo T para o arranjo A, temos $PM(n) = (n)$. Assim, resta apenas mostrar que $PM_1(n) = O(n)$.

Primeiro deduziremos uma recorrência para o trabalho do pior caso. A busca binária na linha 10 custa (lg n) no pior caso, que domina o outro trabalho fora das chamadas recursivas. Para as chamadas recursivas, observe que, se bem que as chamadas nas linhas 13 e 14 poderiam intercalar quantidades diferentes de elementos, as duas chamadas recursivas juntas intercalam no máximo n elementos (na verdade $n - 1$ elementos, visto que $T[q_1]$ não participa de qualquer das duas chamadas recursivas). Além disso, como vimos na análise da duração, uma chamada recursiva funciona para no máximo $3n/4$ elementos. Portanto, obtemos a recorrência

$$PM_1(n) = PM_1(\alpha n) + PM_1((1 - \alpha)n) + O(\lg n), \tag{27.9}$$

onde α encontra-se na faixa $1/4 \le \alpha \le 3/4$ e entendemos que o valor propriamente dito de α pode variar para cada nível de recursão.

Provamos que a recorrência (27.9)tem solução $PM_1 = O(n)$ por meio do método de substituição. Suponha que $PM_1(n) \le c_1 n - c_2 \lg n$ para algumas constantes positivas c_1 e c_2. Substituindo, temos

$$
\begin{aligned}
PM_1(n) &\le (c_1\alpha n - c_2 \lg(\alpha n)) + (c_1(1-\alpha)n - c_2 \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1(\alpha + (1-\alpha))n - c_2(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1 n - c_2(\lg \alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\
&= c_1 n - c_2 \lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\le c_1 n - c_2 \lg n \,,
\end{aligned}
$$

visto que podemos escolher c_2 grande o suficiente para que $c_2(\lg n + \lg(\alpha(1 - \alpha)))$ domine o termo $(\lg n)$. Ainda mais, podemos escolher c_1 grande o suficiente para satisfazer as condições-base da recorrência. Visto que o trabalho $PM_1(n)$ de P-MERGE É (n) e $O(n)$, temos $PM_1(n) = (n)$.

O paralelismo de P-MERGE é $PM_1(n)/PM_\infty(n) = (n/\lg^2 n)$.

Ordenação por intercalação multithread

Agora, que temos um procedimento multithread de intercalação minuciosamente paralelizado, podemos incorporá-lo a uma ordenação por intercalação multithread. Essa versão da ordenação por intercalação é semelhante ao procedimento MERGE-SORT, que já vimos; porém, diferentemente de MERGE-SORT', adota como argumento um subarranjo de saída B, que conterá o resultado ordenado. Em particular, a chamada P-MERGE-SORT(A, p, r, B, s) ordena os elementos em $A[p \,..\, r]$ e os armazena em $B[s \,..\, s + r - p]$.

P-MERGE-SORT(A, p, r, B, s)
1 $\quad n = r - p + 1$
2 $\quad$ **if** $n == 1$
3 $\qquad B[s] = A[p]$
4 $\quad$ **else** seja $T[1 \,..\, n]$ um novo arranjo
5 $\qquad q = \lfloor (p + r)/2 \rfloor$
6 $\qquad q' = q - p + 1$
7 $\qquad$ **spawn** P-MERGE-SORT$(A, p, q, T, 1)$
8 $\qquad$ P-MERGE-SORT$(A, q + 1, r, T, q' + 1)$
9 $\qquad$ **sync**
10 $\qquad$ P-MERGE$(T, 1, q', q' + 1, n, B, s)$

Após a linha 1 calcular o número n de elementos no subarranjo de entrada $A[p \,..\, r]$, as linhas 2–3 tratam o caso-base quando o arranjo tem somente um elemento. As linhas 4–6 se preparam para a geração recursiva na linha 7 e chamam na linha 8, que funciona em paralelo. Em particular, a linha 4 aloca um arranjo temporário T com n elementos para armazenar os resultados da ordenação por intercalação recursiva. A linha 5 calcula o índice q de $A[p \,..\, r]$ para dividir os elementos nos dois subarranjos $A[p \,..\, q]$ e $A[q + 1 \,..\, r]$ que serão ordenados recursivamente, e a linha 6 continua e passa a calcular o número q' de elementos no primeiro subarranjo $A[p \,..\, q]$, que a linha 8 usa para determinar o índice inicial em T referente ao lugar onde será armazenado o resultado ordenado de $A[q + 1 \,..\, r]$. Nesse ponto, a geração e a chamada recursiva são executadas, seguidas pela **sync** na linha 9, que obriga o procedimento a esperar até que o procedimento gerado termine. Finalmente, a linha 10 chama P-MERGE para intercalar os subarranjos ordenados, agora em $T[1 \,..\, q']$ e $T[q' + 1 \,..\, n]$, no subarranjo de saída $B[s \,..\, s + r - p]$.

Análise de ordenação por intercalação multithread

Começamos analisando o trabalho $PMS_1(n)$ de P-MERGE-SORT, o que é consideravelmente mais fácil do que analisar o trabalho de P-MERGE. De fato, o trabalho é dado pela recorrência

$$
\begin{aligned}
PMS_1(n) &= 2\,PMS_1(n/2) + PM_1(n) \\
&= 2\,PMS_1(n/2) + \Theta(n).
\end{aligned}
$$

Essa recorrência é igual à recorrência (4.4) para MERGE-SORT comum dada na Seção 2.3.1 e tem solução $PMS_1(n)$ = $(n \lg n)$ pelo caso 2 do teorema mestre.

Agora, deduzimos e analisamos uma recorrência para a duração do pior caso $PMS(n)$. Como as duas chamadas recursivas a P-MERGE-SORT nas linhas 7 e 8 funcionam logicamente em paralelo, podemos ignorar uma delas, obtendo a recorrência

$$\begin{aligned} PMS_\infty(n) &= PMS_\infty(n/2) + PM_\infty(n) \\ &= PMS_\infty(n/2) + \Theta(\lg^2 n). \end{aligned} \tag{27.10}$$

Como ocorre com a recorrência (27.8), o teorema mestre não se aplica à recorrência (27.10), mas o Exercício 4.6-2 se aplica. A solução é $PMS_1(n) = (\lg_3 n)$ e, assim, a duração de P-MERGE-SORT é $(\lg^3 n)$.

A ordenação paralela dá a P-MERGE-SORT uma vantagem significativa de paralelismo em relação a MERGE-SORT'. Lembre-se de que o paralelismo de MERGE-SORT', que chama o procedimento serial MERGE, é somente $(\lg n)$. Para P-MERGE-SORT, o paralelismo é

$$\begin{aligned} PMS_1(n)/PMS_1(n) &= \Theta(n \lg n)/\Theta(\lg^3 n) \\ &= \Theta(n/\lg^2 n), \end{aligned}$$

que é muito melhor, tanto na teoria quanto na prática. Uma boa implementação na prática sacrificaria algum paralelismo por adensar o caso-base de modo a reduzir as constantes ocultas pela notação assintótica. O modo direto de adensar o caso-base é passar para um tipo de ordenação serial, talvez quicksort, quando o tamanho do arranjo é suficientemente pequeno.

Exercícios

27.3-1 Explique como adensar o caso-base de P-MERGE.

27.3-2 Em vez de encontrar o elemento mediana no subarranjo maior, como faz P-MERGE, considere uma variante que encontra um elemento mediana de todos os elementos nos dois subarranjos ordenados utilizando o resultado do Exercício 9.3-8. Dê pseudocódigo para um procedimento eficiente de intercalação multithread que usa esse procedimento de encontrar o elemento mediana. Analise seu algoritmo.

27.3-3 Dê um algoritmo multithread eficiente para particionar um arranjo em torno de um pivô, como faz o procedimento PARTITION na página 171. Você não precisa particionar o arranjo no lugar. Seu algoritmo deve ser o mais paralelo possível. Analise seu algoritmo.

(*Sugestão:* Talvez você precise de um arranjo auxiliar e também de fazer mais de uma passagem pelos elementos de entrada.)

27.3-4 Dê uma versão multithread de RECURSIVE-FFT na Seção 30.2. Sua implementação deve ser a mais paralela possível. Analise seu algoritmo.

27.3-5 ★ Dê uma versão multithread de RANDOMIZED-SELECT na Seção 9.2. Sua implementação deve ser a mais paralela possível. Analise seu algoritmo. (*Sugestão:* Use o algoritmo de particionamento do Exercício 27.3-3.)

27.3-6 ★ Mostre como aplicar multithread a SELECT da Seção 9.3. Sua implementação deve ser a mais paralela possível. Analise seu algoritmo.

Problemas

27-1 *Implementar laços paralelos utilizando paralelismo aninhado*

Considere o seguinte algoritmo multithread para efetuar adição aos pares em arranjos de n elementos $A[1 \,..\, n]$ e $B[1 \,..\, n]$, armazenando as somas em $C[1 \,..\, n]$.

```
SUM-ARRAYS(A, B, C)
1    parallel for i = 1 to A.comprimento
2        C[i] = A[i] + B[i]
```

a. Reescreva o laço paralelo em SUM-ARRAYS utilizando paralelismo aninhado (**spawn** e **sync**) à maneira de MAT-VEC-MAIN-LOOP. Analise o paralelismo de sua implementação.

Considere a seguinte implementação alternativa do laço paralelo, que contém um valor *granularidade* a ser especificado:

```
SUM-ARRAYS′(A, B, C)
1    n = A.comprimento
2    granularidade = ?                        // a ser determinado
3    r = ⌈n/granularidade⌉
4    for k = 0 to r − 1
5        spawn ADD-SUBARRAY(A, B, C, k · granularidade + 1,
                          min((k + 1) · granularidade, n))
6    sync
```

```
ADD-SUBARRAY(A, B, C, i, j)
1    for k = i to j
2        C[k] = A[k] + B[k]
```

b. Suponha que definimos *granularidade* = 1. Qual é o paralelismo dessa implementação?

c. Dê a fórmula para a duração de SUM-ARRAYS′ em termos de n e *granularidade*. Deduza o melhor valor para *granularidade* para maximizar paralelismo.

27-2 *Economizar espaço temporário em multiplicação de matrizes*

O procedimento P-MATRIX-MULTIPLY-RECURSIVE apresenta a desvantagem de ter de alocar uma matriz temporária T de tamanho $n \times n$, o que pode acarretar efeitos adversos nas constantes ocultas pela notação . Todavia, o procedimento P-MATRIX-MULTIPLY-RECURSIVE realmente tem alto paralelismo. Por exemplo, ignorando as constantes na notação , o paralelismo para multiplicar matrizes 1000×1000 chega a aproximadamente $1000^3/10^2 = 10^7$, visto que $\lg 1000 \approx 10$. A maioria dos computadores paralelos tem muito menos que 10 milhões de processadores.

a. Descreva um algoritmo multithread recursivo que elimine a necessidade da matriz temporária T ao custo de aumentar a duração para (n). (*Sugestão:* Calcule $C = C + AB$ seguindo a estratégia geral de P-MATRIX-MULTIPLY-RECURSIVE, mas inicialize C em paralelo e insira uma **sync** em uma localização criteriosamente escolhida.)

b. Dê e resolva recorrências para o trabalho e duração de sua implementação.

c. Analise o paralelismo de sua implementação. Ignorando as constantes na notação , estime o paralelismo em matrizes 1000 × 1000. Compare com o paralelismo de P-MATRIX-MULTIPLY-RECURSIVE.

27-3 Algoritmos multithread com matrizes

a. Paralelize o procedimento LU-DECOMPOSITION na Seção 28.1 dando pseudocódigo para uma versão multithread desse algoritmo. Sua implementação deve ser a mais paralela possível; analise seu trabalho, duração e paralelismo.

b. Faça o mesmo para LUP-DECOMPOSITION.

c. Faça o mesmo para LUP-SOLVE.

d. Faça o mesmo para um algoritmo multithread baseado na equação (28.13) para inverter uma matriz simétrica definida positiva.

27-4 Multithread aplicado a reduções e cálculos de prefixo

Uma ⊗-redução de um arranjo $x[1\,..\,n]$, onde ⊗ é um operador associativo, é o valor

$$y = x[1] \otimes x[2] \otimes \cdots \otimes x[n].$$

O seguinte procedimento calcula serialmente a ⊗-redução de um subarranjo $x[i\,..\,j]$.

```
REDUCE(x, i, j)
1    y = x[i]
2    for k = i + 1 to j
3        y = y ⊗ x[k]
4    return y
```

a. Use paralelismo aninhado para implementar um algoritmo multithread P-REDUCE, que executa a mesma função com trabalho (n) e duração (lg n). Analise seu algoritmo.

Um problema relacionado é o de calcular uma ***computação ⊗-prefixo***, às vezes denominada ***⊗-scan***, em um arranjo $x[1\,..\,n]$, onde ⊗ é novamente um operador associativo. A ⊗-scan produz o arranjo $[1\,..\,n]$ dado por

$$\begin{aligned} y[1] &= x[1]\,, \\ y[2] &= x[1] \otimes x[2]\,, \\ y[3] &= x[1] \otimes x[2] \otimes x[3]\,, \\ &\vdots \\ y[n] &= x[1] \otimes x[2] \otimes x[3] \otimes \cdots \otimes x[n]\,, \end{aligned}$$

isto é, todos os prefixos do arranjo x “somados” utilizando o operador ⊗. O seguinte procedimento serial SCAN executa uma computação ⊗-prefixo:

```
SCAN(x)
1    n = x.comprimento
2    seja y[1 . . n] um novo arranjo
3    y[1] = x[1]
4    for i = 2 to n
5        y[i] = y[i − 1] ⊗ x[i]
6    return y
```

Infelizmente, aplicar multithread a SCAN não é direto. Por exemplo, mudar o laço **for** para um laço **parallel for** criaria corridas, visto que cada iteração do corpo do laço depende da iteração anterior. O seguinte procedimento P-SCAN-1 executa a computação prefixo ⊗ em paralelo, se bem que ineficientemente:

```
P-SCAN-1(x)
1 n = x.comprimento
2 seja y[1 . . n] um novo arranjo
3 P-SCAN-1-AUX(x, y, 1, n)
4 return y
```

```
P-SCAN-1-AUX(x, y, i, j)
1 parallel for l = i to j
2        y[l] = P-REDUCE(x, 1, l)
```

b. Analise o trabalho, duração e paralelismo de P-SCAN-1.

Utilizando paralelismo aninhado, podemos obter uma computação ⊗-prefixo mais eficiente:

```
P-SCAN-2(x)
1    n = x.comprimento
2    seja y[1 . . n] um novo arranjo
3    P-SCAN-2-AUX(x, y, 1, n)
4    return y
```

```
P-SCAN-2-AUX(x, y, i, j)
1    if i == j
2        y[i] = x[i]
3    else k = ⌊(i + j)/2⌋
4        spawn P-SCAN-2-AUX(x, y, i, k)
5        P-SCAN-2-AUX(x, y, k + 1, j)
6        sync
7        parallel for l = k + 1 to j
8            y[l] = y[k] ⊗ y[l]
```

c. Demonstre que P-SCAN-2 é correto e analise seu trabalho, duração e paralelismo.

Podemos melhorar P-SCAN-1 e P-SCAN-2 executando a computação prefixo ⊗ em duas passagens distintas pelos dados. Na primeira passagem, reunimos os termos para vários subarranjos contíguos de x em um arranjo temporário t, e na segunda passagem usamos os termos em t para calcular o resultado final y. O seguinte pseudocódigo implementa essa estratégia, mas certas expressões foram omitidas:

```
P-SCAN-3(x)
1    n = x.comprimento
2    sejam y[1 . . n] e t[1 . . n] novos arranjos
3    y[1] = x[1]
4    if n > 1
5        P-SCAN-UP(x, t, 2, n)
6        P-SCAN-DOWN(x[1], x, t, y, 2, n)
7    return y
```

```
P-SCAN-UP(x, t, i, j)
1    if i == j
2        return x[i]
3    else
4        k = ⌊(i + j)/2
5        t [k] = spawn P-SCAN-UP(x, t, i, k)
6        right = P-SCAN-UP(x, t, k + 1, j)
7        sync
8        return __________              // preencha a lacuna
```

```
P-SCAN-DOWN(v, x, t, y, i, j)
1    if i == j
2        y[i] = v ⊗ x[i]
3    else
4        k = ⌊(i + j)/2⌋
5        spawn P-SCAN-DOWN(______ , x, t, y, i, k)      // preencha a lacuna
6        P-SCAN-DOWN(______, x, t, y, k + 1, j)         // preencha a lacuna
7        sync
```

d. Preencha as três expressões que faltam na linha 8 de P-SCAN-UP e nas linhas 5 e 6 de P-SCAN-DOWN. Demonstre que, com as expressões que você deu, P-SCAN-3 é correto. (*Sugestão:* Prove que o valor v passado para P-SCAN-DOWN(v, x, t, y, i, j) satisfaz $v = x[1] \otimes x[2] \otimes \ldots \otimes x[i-1]$.)

e. Analise o trabalho, a duração e o paralelismo de P-SCAN-3.

27-5 Multithread aplicado a um cálculo simples com estêncil

A ciência da computação está repleta de algoritmos que exigem que as entradas de um arranjo sejam preenchidas com valores que dependem dos valores já calculados de certas entradas vizinhas, juntamente com outras informações que não mudam no curso da computação. O padrão de entradas vizinhas não muda durante a computação e é denominado ***estêncil***. Por exemplo, a Seção 15.4 apresenta um algoritmo estêncil para calcular uma subsequência comum mais longa, onde o valor na entrada $c[i, j]$ depende somente dos valores em $c[i-1, j]$, $c[i, j-1]$, e $c[i-1, j-1]$, bem como dos elementos x_i e y_j dentro das duas sequências dadas como entradas. As sequências de entrada são fixas, mas o algoritmo preenche o arranjo bidimensional

c, de modo a calcular a entrada $c[i, j]$ após calcular todas as três entradas $c[i–1, j]$, $c[i, j –1]$, e $c[i – 1, j – 1]$.

Nesse problema, examinamos como usar paralelismo aninhado para aplicar multithread a cálculo simples com estêncil em um arranjo $n \times n$ A, no qual, dos valores em A, o valor colocado dentro da entrada $A[i, j]$ depende somente dos valores em $A[i', j']$, onde $i' \leq i$ e $j' \leq j$ (e, é claro, $i' \neq i$ ou $j' \neq j$). Em outras palavras, o valor em uma entrada depende somente dos valores em entradas que estão acima dele ou à sua esquerda, juntamente com a informação estática que está fora do arranjo. Além disso, supomos em todo esse problema que, uma vez preenchidas as entradas das quais $A[i, j]$ depende, podemos preencher $A[i, j]$ no tempo (1) (como no procedimento LCS-Length da Seção 15.4).

Podemos particionar o arranjo $n \times n$ A em quatro subarranjos $n/2 \times n/2$ da seguinte maneira:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}. \qquad (27.11)$$

Observe agora que podemos preencher o subarranjo A_{11} recursivamente, visto que ele não depende das entradas dos outros três subarranjos. Uma vez completado A_{11}, podemos continuar a preencher A_{12} e A_{21} recursivamente em paralelo porque, embora ambos dependam de A_{11}, não dependem um do outro. Por fim, podemos preencher A_{22} recursivamente.

a. Dê pseudocódigo multithread que execute esses cálculo simples com estêncil utilizando um algoritmo de dividir e conquistar Simple-Stencil baseado na decomposição (27.11) e na discussão anterior. (Não se preocupe com os detalhes do caso-base, que depende de cada estêncil específico.) Dê e resolva recorrências para o trabalho e a duração desse algoritmo em termos de n. Qual é o paralelismo?

b. Modifique sua solução para a parte (a) para dividir um arranjo $n \times n$ em nove subarranjos $n/3 \times n/3$, novamente executando recursão com o máximo de paralelismo possível. Analise esse algoritmo. Mais ou menos quanto paralelismo tem esse algoritmo em comparação com o algoritmo da parte (a)?

c. Generalize suas soluções para as partes (a) e (b) da seguinte maneira: escolha um inteiro $b \geq 2$. Divida um arranjo $n \times n$ em b_2 subarranjos, cada um de tamanho $n/b \times n/b$, executando recursão com o máximo de paralelismo possível. Em termos de n e b, quais são o trabalho, a duração e o paralelismo de seu algoritmo? Demonstre que, utilizando essa abordagem, o paralelismo deve ser $o(n)$ para qualquer escolha de $b \geq 2$. (*Sugestão:* Para este último argumento, mostre que o expoente de n no paralelismo é estritamente menor que 1 para qualquer escolha de $b \geq 2$.)

d. Dê pseudocódigo para um algoritmo multithread para esse cálculo simples com estêncil que consiga paralelismo $(n/\lg n)$. Demonstre, utilizando noções de trabalho e duração, que, na verdade, o problema tem paralelismo inerente (n). Acontece que a natureza de divisão e conquista do nosso pseudocódigo multithread não nos permite conseguir esse paralelismo máximo.

27-6 Algoritmos multithread aleatorizados

Exatamente como ocorre com os algoritmos seriais comuns, às vezes, queremos implementar algoritmos multithread aleatorizados. Este problema explora como adaptar as várias medições de desempenho para tratar o comportamento esperado de tais algoritmos. O problema pede também que você projete e analise um algoritmo multithread para quicksort aleatorizada.

a. Explique como modificar a lei do trabalho (27.2), a lei da duração (27.3) e o limite do escalonador guloso (27.4) para trabalhar com esperanças quando T_P, T_1 e T_∞ são variáveis aleatórias.

b. Considere um algoritmo multithread aleatorizado para o qual durante 1% do tempo temos $T_1 = 10^4$ e $T_{10.000} = 1$, mas durante 99% do tempo temos $T_1 = T_{10.000} = 10^9$. Explique por que o ***fator de aceleração*** do algoritmo multithread aleatorizado deve ser definido como $E[T_1] / E[T_P]$, em vez de $E[T_1/T_P]$.

c. Explique por que o ***paralelismo*** de um algoritmo multithread aleatorizado deve ser definido como a razão $E\ T_1/E\ T_\infty$.

d. Aplique multithread ao algoritmo RANDOMIZED-QUICKSORT na página 179 utilizando paralelismo aninhado. (Não paralelize RANDOMIZED-PARTITION.) Dê o pseudocódigo para seu algoritmo P-RANDOMIZED-QUICKSORT.

e. Analise seu algoritmo multithread para quicksort aleatorizada. (*Sugestão:* Revise a análise de RANDOMIZED-SELECT na Seção 9.2.)

NOTAS DO CAPÍTULO

Computadores paralelos, modelos para computadores paralelos e modelos algorítmicos para programação paralela estão por aí há anos, sob várias formas. Edições anteriores deste livro incluíam material sobre redes de ordenação e sobre o modelo PRAM (Parallel Random-Access Machine — Máquina Paralela de Acesso Aleatório). O modelo de dados paralelos [48, 168] é um outro modelo popular de programação algorítmica que apresenta operações com vetores e matrizes como primitivas.

Graham [149] e Brent [55] mostraram que existem escalonadores que alcançam os limites do Teorema 27.1. Blumofe e Leiserson [52] mostraram que todo escalonador guloso atinge o limite. Blelloch [47] desenvolveu um modelo de programação algorítmica baseado em trabalho e duração (que ele denominou "profundidade" da computação) para programação de dados paralelos. Blumofe e Leiserson [52] deram um algoritmo de escalonamento distribuído para multithread dinâmico baseado em "roubo de trabalho" aleatorizado e mostraram que ele alcança o limite $E[T_P] \leq T_1/P + O(T_\infty)$. Arora, Blumofe e Plaxton [19] e Blelloch, Gibbons e Matias [49] também apresentaram algoritmos provadamente bons para escalonar computações multithread dinâmicas.

O modelo de pseudocódigo e programação multithread foi muito influenciado pelo projeto Cilk [51, 118] do MIT e pelas extensões Cilk++ [71] de C++ distribuídas por Cilk Arts, Inc. Muitos dos algoritmos multithread neste capítulo apareceram em notas não publicadas de conferências realizadas por C. E. Leiserson e H. Prokop, e foram implementadas em Cilk ou Cilk++. O algoritmo multithread de ordenação por intercalação foi inspirado num algoritmo de Akl [12].

A noção de consistência sequencial se deve a Lamport [223].

---

* N. do RT: Embora algumas traduções para "threads" sejam correntes, preferimos importar a palavra diretamente; somente ela e não suas flexões.

28 OPERAÇÕES COM MATRIZES

Como operações com matrizes encontram-se no núcleo da computação científica, algoritmos eficientes para trabalhar com matrizes têm muitas aplicações práticas. Este capítulo focaliza como multiplicar matrizes e resolver conjuntos de equações lineares simultâneas. O Apêndice D dá uma revisão dos fundamentos de matrizes.

A Seção 28.1 mostra como resolver um conjunto de equações lineares usando decomposições LUP. Em seguida, a Seção 28.2 explora a estreita relação entre multiplicar e inverter matrizes. Finalmente, a Seção 28.3 a importante classe de matrizes simétricas positivas definidas e mostra como podemos usá-las para determinar uma solução de mínimos quadrados para um conjunto superdeterminado de equações lineares.

Uma questão importante que surge na prática é a ***estabilidade numérica***. Devido à precisão limitada de representações de ponto flutuante em computadores reais, erros de arredondamento em computações numéricas podem ser amplificados no curso de uma computação, levando a resultados incorretos; tais computações são denominadas ***numericamente instáveis***. Se bem que faremos breve menção à estabilidade numérica, não a abordaremos neste capítulo. Para uma discussão completa de questões de estabilidade, indicamos o excelente livro de Golub e Van Loan [144].

28.1 RESOLVENDO SISTEMAS DE EQUAÇÕES LINEARES

Numerosas aplicações precisam resolver conjuntos de equações lineares simultâneas. Podemos formular um sistema linear como uma equação matricial na qual cada elemento de matriz ou de vetor pertence a um corpo, normalmente o dos números reais . Esta seção discute como resolver um sistema de equações lineares usando um método denominado decomposição LUP.

Começamos com um conjunto de equações lineares com n incógnitas $x_1, x_2, ..., x_n$:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &= b_1 , \\
a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n &= b_2 , \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n &= b_n .
\end{aligned} \tag{28.1}$$

Uma ***solução*** para as equações (28.1) é um conjunto de valores para $x_1, x_2, \ldots, x_n$ que satisfaz todas as equações simultaneamente. Nesta seção trataremos somente do caso em que há exatamente n equações com n incógnitas.

Um modo conveniente de expressar as equações (28.1) é sob a forma de uma equação matricial de vetores

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

ou, o que é equivalente, escrever $A = (a_{ij})$, $x = (x_i)$ e $b = (b_i)$ como

$$Ax = b \,. \tag{28.2}$$

Se A é não singular, possui uma inversa A^{-1}, e

$$x = A^{-1}b \tag{28.3}$$

é o vetor solução. Podemos provar que x é a única solução para a equação (28.2) da seguinte maneira: se existem duas soluções, x e x', então $Ax = Ax' = b$ e, denotando uma matriz identidade por I,

$$\begin{aligned} x &= Ix \\ &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}A)x' \\ &= x'. \end{aligned}$$

Nesta seção, nossa preocupação predominante será o caso em que A é não singular ou, o que é equivalente (pelo Teorema D.1), o posto de A é igual ao número n de incógnitas. Contudo, existem outras possibilidades que merecem uma breve discussão. Se o número de equações é menor que o número n de incógnitas ou, de modo mais geral, se o posto de A é menor que n, o sistema é ***subdeterminado***. Normalmente, um sistema subdeterminado tem um número infinito de soluções, embora possa não ter absolutamente nenhuma solução, se as equações são inconsistentes. Se o número de equações for maior que o número n de incógnitas, o sistema é ***superdeterminado***, e é possível que não exista nenhuma solução. A Seção 28.3 aborda o importante problema de determinar boas soluções aproximadas para sistemas superdeterminados de equações lineares.

Vamos retornar ao nosso problema de resolver o sistema $Ax = b$ de n equações com n incógnitas. Poderíamos calcular A^{-1} e, depois, usando a equação (28.3), multiplicar b por A^{-1} e obter $x = A^{-1}b$. Na prática essa abordagem sofre de instabilidade numérica. Felizmente existe outra abordagem — a decomposição LUP —, que é numericamente estável e tem a vantagem adicional de ser mais rápida na prática.

Visão geral da decomposição LUP

A ideia por trás da decomposição LUP é encontrar três matrizes $n \times n$, L, U e P, tais que

$$PA = LU \,, \tag{28.4}$$

onde

- L é uma matriz triangular inferior unitária,
- U é uma matriz triangular superior e
- P é uma matriz de permutação.

Denominamos as matrizes L, U e P que satisfazem a equação (28.4) ***decomposição LUP*** da matriz A. Mostraremos que toda matriz não singular A possui tal decomposição.

A vantagem de calcular uma decomposição LUP para a matriz A é que é mais fácil resolver sistemas lineares quando as matrizes são triangulares, como é o caso das matrizes L e U. Uma vez determinada uma decomposição LUP para A, podemos resolver a equação (28.2) $Ax = b$ resolvendo somente sistemas lineares triangulares da maneira mostrada a seguir. Multiplicando ambos os lados de $Ax = b$ por P obtemos a equação equivalente $PAx = Pb$ que, pelo Exercício D.1-4, equivale a permutar as equações (28.1). Usando nossa decomposição (28.4), obtemos

$$LUx = Pb .$$

Agora podemos resolver essa equação resolvendo dois sistemas lineares triangulares. Definimos $y = Ux$, onde x é o vetor solução desejado. Primeiro, resolvemos o sistema triangular inferior

$$Ly = Pb \tag{28.5}$$

para o vetor incógnita y por um método denominado "substituição direta". Depois de resolvido para y, resolvemos o sistema triangular superior

$$Ux = y \tag{28.6}$$

para a incógnita x por um método denominado "substituição reversa". Como a matriz de permutação P é inversível (Exercício D.2-3), multiplicamos ambos os lados da equação (28.4) por P^{-1} dá $P^{-1}PA = P^{-1}LU$, de modo que

$$A = P^{-1}LU. \tag{28.7}$$

Por consequência, o vetor x é a nossa solução para $Ax = b$:

$$\begin{aligned} Ax &= P^{-1}LUx && \text{(pela equação (28.7))} \\ &= P^{-1}Ly && \text{(pela equação (28.6))} \\ &= P^{-1}Pb && \text{(pela equação (28.5))} \\ &= b. \end{aligned}$$

Nossa próxima etapa é mostrar como funcionam a substituição direta e a substituição reversa e atacar o problema do cálculo da decomposição LUP.

Substituição direta e inversa

Substituição direta pode resolver o sistema triangular inferior (28.5) no tempo (n_2), dados L, P e b. Por conveniência, representamos a permutação P compactamente por um arranjo $[1 .. n]$. Para $i = 1, 2, ..., n$, a entrada $p[i]$ indica que $P_{i, p[i]} = 1$ e $P_{ij} = 0$ para $j \neq p[i]$. Assim, PA tem $a_{p[i]}$ na linha i e coluna j, e Pb tem $b_{p[i]}$ como seu i-ésimo elemento. Visto que L é triangular inferior unitária, a equação (28.5) pode ser reescrita como

$$\begin{aligned}
y_1 &= b_{\pi[1]}, \\
l_{21}y_1 + \quad y_2 &= b_{\pi[2]}, \\
l_{31}y_1 + l_{32}y_2 + \quad y_3 &= b_{\pi[3]}, \\
&\vdots \\
l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]}.
\end{aligned}$$

A primeira equação nos diz que $y_1 = b_{p[1]}$. Conhecendo o valor de y_1 podemos substituí-lo na segunda equação, o que dá

$$y_2 = b_{\pi[2]} - l_{21}y_1 \,.$$

Agora, podemos substituir y_1 e y_2 na terceira equação, obtendo

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2) \,.$$

Em geral, substituímos $y_1, y_2, ..., y_{i-1}$ "diretamente" na i-ésima equação para resolver para y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j \,.$$

Agora, que resolvemos para y, resolvemos para x na equação (28.6) usando ***substituição reversa***, que é semelhante à substituição direta. Aqui, resolvemos primeiro a n-ésima equação e trabalhamos em sentido contrário até a primeira equação. Como a substituição direta, esse processo é executado no tempo (n_2). Visto que U é triangular superior, podemos reescrever o sistema (28.6) como

$$\begin{aligned}
u_{11}x_1 + u_{12}x_2 + \cdots + \quad u_{1,n-2}x_{n-2} + \quad u_{1,n-1}x_{n-1} + \quad u_{1n}x_n &= y_1 \,, \\
u_{22}x_2 + \cdots + \quad u_{2,n-2}x_{n-2} + \quad u_{2,n-1}x_{n-1} + \quad u_{2n}x_n &= y_2 \,, \\
&\vdots \\
u_{n-2,n-2}x_{n-2} + \quad u_{n-2,n-1}x_{n-1} + \quad u_{n-2,n}x_n &= y_{n-2} \,, \\
u_{n-1,n-1}x_{n-1} + \quad u_{n-1,n}x_n &= y_{n-1} \,, \\
u_{n,n}x_n &= y_n \,.
\end{aligned}$$

Assim, podemos resolver sucessivamente para x_n, x_n-1, ..., x_1 , da seguinte maneira:

$$\begin{aligned}
x_n &= y_n/u_{n,n} \,, \\
x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \,, \\
x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2},{}_nx_n))/u_{n-2,n-2} \,, \\
&\vdots
\end{aligned}$$

ou, em geral,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii}.$$

Dados P, L, U e b, o procedimento LUP-Solve resolve para x combinando substituição direta e substituição reversa. O pseudocódigo considera que a dimensão n aparece no atributo $L.linhas$ e que a matriz de permutação P é representada pelo arranjo p.

LUP-Solve(L, U, π, b)
1 $n = L.linhas$
2 sejam x e y novos vetores de comprimento n
3 **for** $i = 1$ **to** n
4 $y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j$
5 **for** $i = n$ **downto** 1
6 $x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii}$
7 **return** x

O procedimento LUP-Solve resolve para y usando substituição direta nas linhas 3-4 e depois resolve para x usando substituição inversa nas linhas 5-6. Visto que o somatório dentro de cada um dos laços **for** inclui um laço implícito, o tempo de execução é (n_2).

Como exemplo desses métodos, considere o sistema de equações lineares definido por

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix},$$

onde

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix},$$

$$b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix}$$

que desejamos resolver para a incógnita x. A decomposição LUP é

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0{,}2 & 1 & 0 \\ 0{,}6 & 0{,}5 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0{,}8 & -0{,}6 \\ 0 & 0 & 2{,}5 \end{pmatrix},$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

(Seria interessante você verificar que $PA = LU$.) Usando substituição direta, resolvemos $Ly = Pb$ para y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0{,}2 & 1 & 0 \\ 0{,}6 & 0{,}5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix},$$

e obtemos

$$y = \begin{pmatrix} 8 \\ 1{,}4 \\ 1{,}5 \end{pmatrix}$$

calculando primeiro y_1, depois y_2 e, finalmente, y_3. Usando substituição inversa, resolvemos $Ux = y$ para x:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0,8 & -0,6 \\ 0 & 0 & 2,5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1,4 \\ 1,5 \end{pmatrix},$$

obtendo assim a resposta desejada

$$x = \begin{pmatrix} -1,4 \\ 2,2 \\ 0,6 \end{pmatrix}$$

calculando primeiro x_3, depois x_2 e finalmente x_1.

Calculando uma decomposição LU

Agora já mostramos que, se podemos criar uma decomposição LUP para uma matriz não singular A, substituição direta e substituição inversa podem resolver o sistema $Ax = b$ de equações lineares. Agora mostraremos como calcular eficientemente uma decomposição LUP para A. Começamos com o caso no qual A é uma matriz não singular $n \times n$ e P está ausente (ou, o que é equivalente, $P = I_n$). Nesse caso, fatoramos $A = LU$. Denominamos as duas matrizes L e U ***decomposição LU*** de A. Usamos um processo conhecido como ***método de eliminação de Gauss*** para criar uma decomposição LU. Começamos subtraindo múltiplos da primeira equação das outras equações para eliminar a primeira variável dessas equações. Então, subtraímos múltiplos da segunda equação da terceira equação e das equações subsequentes, de modo que agora a primeira e a segunda variáveis são eliminadas dessas equações. Continuamos esse processo até que o sistema remanescente tenha forma triangular superior — na verdade, ele é a matriz U. A matriz L é formada pelos multiplicadores de linha que provocam a eliminação de variáveis.

Nosso algoritmo para implementar essa estratégia é recursivo. Desejamos construir uma decomposição LU para uma matriz não singular $n \times n$ A. Se $n = 1$, terminamos, já que podemos escolher $L = I_1$ e $U = A$. Para $n > 1$, dividimos A em quatro partes:

$$A = \left(\begin{array}{c|cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) = \begin{pmatrix} a_{11} & w^{T} \\ v & A' \end{pmatrix},$$

onde $v = (v_2, v_3, ..., v_n) = (a_{21}, a_{31}, ..., a_n 1)$ é um $(n - 1)$-vetor coluna, $w^T = (w_2, w_3, ..., w_n)^T = (a_{12}, a_{13}, ..., a_{1n})^T$ é um $(n - 1)$-vetor linha e A' é uma matriz $(n - 1) \times (n - 1)$. Então, usando álgebra de matrizes (verifique as equações simplesmente efetuando as multiplicações), podemos fatorar A como

$$A = \begin{pmatrix} a_{11} & w^{T} \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{T} \\ 0 & A' - vw^{T}/a_{11} \end{pmatrix}. \tag{28.8}$$

Os zeros na primeira e na segunda matrizes da equação (28.8) são (n - 1)-vetores linha e coluna, respectivamente. O termo vw_T/a_{11}, formado tomando o produto externo de v e w e dividindo cada elemento do resultado por a_{11}, é uma matriz (n - 1) × (n - 1), que corresponde em tamanho à matriz A da qual ela é subtraída. A matriz (n - 1) × (n - 1) resultante

$$A' - vw^{T}/a_{11} \tag{28.9}$$

é denominada ***complemento de Schur*** de A em relação a a_{11}.

Afirmamos que, se A é não singular, o complemento de Schur também é não singular. Por quê? Suponha que o complemento de Schur, que é (n - 1) × (n - 1), seja singular. Então, pelo Teorema D.1, ele tem posto linha estritamente menor que n - 1. Como as n - 1 entradas inferiores na primeira coluna da matriz

$$\begin{pmatrix} a_{11} & w^{T} \\ 0 & A' - vw^{T}/a_{11} \end{pmatrix}$$

são zero, as n - 1 linhas inferiores dessa matriz devem ter posto estritamente menor que n - 1. Portanto, o posto linha da matriz inteira é estritamente menor que n. Aplicando o Exercício D.2-8 à equação (28.8), A tem posto estritamente menor que n e, pelo Teorema D.1, deduzimos a contradição de que A é singular.

Como o complemento de Schur é não singular, agora podemos determinar recursivamente uma decomposição LU para ele. Digamos que

$$A' - vw^{T}/a_{11} = L'U' ,$$

onde L' é triangular inferior unitária e U' é triangular superior. Então, usando álgebra de matrizes, temos

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{T} \\ 0 & A' - vw^{T}/a_{11} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{T} \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{T} \\ 0 & U' \end{pmatrix} \\ &= LU, \end{aligned}$$

o que dá a nossa decomposição LU. (Observe que, como L' é uma matriz triangular inferior unitária, L também é, e como U' é triangular superior, U também é.)

É claro que se a_{11} = 0, esse método não funciona porque divide por zero. Também não funciona se a entrada superior da extrema esquerda do complemento de Schur A' - vw_T/a_{11} para zero, já que dividimos por ela na etapa seguinte da recursão. Os elementos que usamos como divisores durante a decomposição LU são denominados ***pivôs*** e

ocupam os elementos diagonais da matriz U. A razão por que incluímos uma matriz permutação P durante a decomposição LUP é que ela nos permite evitar a divisão por zero. Quando usamos permutações para evitar a divisão por zero (ou por números pequenos) estamos ***pivotando.***

Uma classe importante de matrizes para as quais a decomposição LU sempre funciona corretamente é a classe das matrizes simétricas positivas definidas. Tais matrizes não exigem pivotamento e, assim, podemos empregar a estratégia recursiva que acabamos de delinear sem medo de dividir por zero. Provaremos esse resultado, bem como vários outros, na Seção 28.3.

Nosso código para decomposição LU de uma matriz A segue a estratégia recursiva, exceto que um laço de iteração substitui a recursão. (Essa transformação é uma otimização-padrão para um procedimento com "recursão de cauda", um procedimento cuja última operação é uma chamada recursiva a ele mesmo. Veja o Problema 7.4.) O código supõe que o atributo $A.linhas$ dá a dimensão de A. Inicializamos a matriz U com zeros abaixo da diagonal e a matriz L com 1s em sua diagonal e zeros acima da diagonal. Cada iteração trabalha com uma submatriz quadrada, usando o elemento do canto superior esquerdo como pivô para computar os vetores v e w e o complemento de Schur, que passa a ser a matriz quadrada com que a próxima iteração trabalha.

```
LU-DECOMPOSITION(A)
1   n = A.linhas
2   sejam L e U as novas matrizes n × n
3   inicialize U com zeros abaixo da diagonal
4   inicialize L com 1s na diagonal e zeros acima da diagonal
5   for k = 1 to n
6        u_kk = a_kk
7        for i = k + 1 to n
8             l_ik = a_ik/a_kk      // l_ik contém v_i
9             u_ki = a_ki           // u_ki contém w_i^T
10       for i = k + 1 to n
11            for j = k + 1 to n
12                 a_ij = a_ij − l_ik u_kj
13  return L and U
```

O laço **for** externo, que começa na linha 5, itera uma vez para cada etapa recursiva. Dentro desse laço, a linha 6 determina que o pivô é $u_{kk} = a_{kk}$. O laço **for** nas linhas 7-9 (que não é executado quando $k = n$), usa os vetores v e w para atualizar L e U. A linha 8 determina os elementos de L abaixo da diagonal, armazenando v_i/a_{kk} em l_{ik}, e a linha 9 calcula os elementos de L acima da diagonal, armazenando w_i em u_{ki}. Finalmente, as linhas 10-12 calculam os elementos do complemento de Schur e os armazenam de volta na matriz A (Não precisamos dividir por a_{kk} na linha 12 porque já o fizemos quando calculamos l_{ik} na linha 8.) Como a linha 12 é triplamente aninhada, LU-DECOMPOSITION é executada no tempo (n_3).

A Figura 28.1 ilustra a operação de LU-DECOMPOSITION. Ela mostra uma otimização-padrão do procedimento, na qual armazenamos os elementos significativos de L e U no lugar na matriz A. Isto é, podemos configurar uma correspondência entre cada elemento a_{ij} e ou l_{ij} (se $i > j$) ou u_{ij} (se $i \leq j$) e atualizar a matriz A para que ela contenha L e U quando o procedimento termina. Para obter o pseudocódigo para essa otimização pelo pseudocódigo dado, basta substituir cada referência a l ou u por a; é fácil verificar que essa transformação preserva a correção.

Calculando uma decomposição LUP

Em geral, quando resolvemos um sistema de equações lineares $Ax = b$, temos de pivotar em elementos de A que estão fora da diagonal para evitar divisão por zero. Claro que dividir por zero seria desastroso. Porém, também queremos evitar dividir por um valor pequeno mesmo que A seja não singular — porque isso pode produzir instabilidades numéricas. Então, tentamos pivotar em um valor grande.

A matemática por trás da decomposição LUP é semelhante à da decomposição LU. Lembre-se de que temos uma matriz $n \times n$ não singular A e desejamos encontrar uma matriz de permutação P, uma matriz triangular inferior unitária L

e uma matriz triangular superior U, tais que $PA = LU$. Antes de particionar a matriz A, como fizemos para a decomposição LU, passamos um elemento não nulo; digamos a_{k1}, de algum lugar na primeira coluna até a posição (1, 1) da matriz. Para estabilidade numérica, escolhemos a_{k1} como o elemento na primeira coluna que tem o maior valor absoluto. (A primeira coluna não pode conter somente zeros porque A seria singular, já que seu determinante seria zero pelos Teoremas D.4 e D.5.) Para preservar o conjunto de equações, trocamos a linha 1 com a linha k, o que equivale a multiplicar A por uma matriz de permutação Q à esquerda (Exercício D.14). Assim, podemos escrever QA como

$$QA = \begin{pmatrix} a_{k1} & w^T \\ v & A' \end{pmatrix}.$$

$$\text{(a)}\quad \begin{matrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{matrix} \qquad \text{(b)}\quad \begin{matrix} \textcircled{2} & 3 & 1 & 5 \\ 3 & 4 & 2 & 4 \\ 1 & 16 & 9 & 18 \\ 2 & 4 & 9 & 21 \end{matrix} \qquad \text{(c)}\quad \begin{matrix} 2 & 3 & 1 & 5 \\ 3 & \textcircled{4} & 2 & 4 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 7 & 17 \end{matrix} \qquad \text{(d)}\quad \begin{matrix} 2 & 3 & 1 & 5 \\ 3 & 4 & 2 & 4 \\ 1 & 4 & \textcircled{1} & 2 \\ 2 & 1 & 7 & 3 \end{matrix}$$

$$\underset{A}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix}} = \underset{L}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix}} \underset{U}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}}$$

(e)

Figura 28.1 A operação de LU-DECOMPOSITION. (**a**) A matriz A. (**b**) O elemento $a_{11} = 2$ no círculo preto é o pivô, a coluna sombreada é v/a e a linha sombreada é w^T. Os elementos de U calculados até agora estão acima da linha horizontal, e os elementos de L estão à esquerda da linha vertical. A matriz complemento de Schur $A' - vw^T/a_{11}$ ocupa a parte inferior direita. (**c**) Agora operamos na matriz complemento de Schur produzida pela parte (b). O elemento $a_{22} = 4$ no círculo preto é o pivô, e a coluna e a linha sombreadas são v/a_{22} e w^T (no particionamento do complemento de Schur), respectivamente. As linhas dividem a matriz nos elementos de U calculados até agora (acima), elementos de L calculados até agora (esquerda) e o novo complemento de Schur (direita inferior). (**d**) Após a próxima etapa, a matriz A está fatorada. (O elemento 3 no novo complemento de Schur se torna parte de U quando a recursão termina.) (**e**) Fatoração $A = LU$.

---

onde $v = (a_{21}, a_{31}, ..., a_{n1})$, exceto que a_{11} substitui a_{k1}; $w = (a_{k2}, a_{k3}, ..., a_{kn})$; e A é uma matriz $(n - 1) \times (n - 1)$. Visto que a_{k1} 0, agora podemos executar praticamente a mesma álgebra linear que usamos para a decomposição LU, porém, desta vez, garantindo que não haverá divisão por zero:

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & w^T \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix}. \end{aligned}$$

Como vimos na decomposição LU, se A é não singular, o complemento de Schur $A = vw_T/a$ também é não singular. Portanto, podemos determinar recursivamente uma decomposição LUP para ela com a matriz triangular inferior unitária L, a matriz triangular superior U e a matriz de permutação P, tais que

$$P'(A' - vw^{\mathrm{T}}/a_{k1}) = L'U'.$$

Defina

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q,$$

que é uma matriz de permutação, visto que é o produto de duas matrizes permutação (Exercício D.1-4). Agora temos

$$\begin{aligned}
PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\
&= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & P'(A' - vw^T/a_{k1}) \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & L'U' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & U' \end{pmatrix} \\
&= LU,
\end{aligned}$$

o que produz a decomposição LUP. Como L é triangular inferior unitária, L também é, e como U é triangular superior, U também é.

Observe que, nessa dedução, diferentemente daquela que ocorre na decomposição LU, temos de multiplicar o vetor coluna $v/a_k{}^1$ e o complemento de Schur $A' - vw_T/a_k{}^1$ pela matriz de permutação P'. Apresentamos a seguir o pseudocódigo para decomposição LUP:

```
LUP-composition(A)
1   n = A.linhas
2   seja π[1 . . n] um novo arranjo
3   for i = 1 to n
4       π[i] = i
5   for k = 1 to n
6       p = 0
7       for i = k to n
8           if |a_ik| > p
9               p = |a_ik|
10              k' = i
11      if p == 0
12          error "matriz singular"
13      trocar π[k] por π[k']
14      for i = 1 to n
15          trocar a_ki por a_k'i
16      for i = k + 1 to n
17          a_ik = a_ik/a_kk
18          for j = k + 1 to n
19              a_ij = a_ij − a_ik a_kj
```

Como LU-Decomposition, nosso pseudocódigo para decomposição LUP substitui a recursão por um laço de iteração. Como uma melhoria em relação à implementação direta da recursão, mantemos dinamicamente a matriz de permutação P como um arranjo p, onde $p[i] = j$ significa que a i-ésima linha de P contém 1 na coluna j. Também implementamos o código para calcular L e U "no lugar" na matriz A. Portanto, quando o procedimento termina,

$$a_{ij} = \begin{cases} l_{ij} & \text{se } i \geq j, \\ u_{ij} & \text{se } i \leq j. \end{cases}$$

A Figura 28.2 ilustra como LUP-Decomposition fatora uma matriz. As linhas 3-4 inicializam o arranjo p para representar a permutação identidade. O laço **for** externo que começa na linha 5 implementa a recursão. Cada vez que passamos pelo laço externo, as linhas 6-10 determinam o elemento a_{kk} com maior valor absoluto dentre os que estão na primeira coluna atual (a coluna k) da matriz $(n - k + 1) \times (n - k + 1)$ cuja decomposição LU estamos determinando. Se todos os elementos na primeira coluna atual são zero, as linhas 11-12 informam que a matriz é singular. Para pivotar, trocamos $p[k']$ por $p[k]$ na linha 13 e trocamos a k-ésima e a k -ésima linha de A nas linhas 14-15, o que transforma o elemento a_{kk} em pivô. (As linhas inteiras são trocadas porque, na dedução do método anterior, não é só $A = vw_{T/}a_k{}^1$ que é multiplicado por P; $v/a_k{}^1$ também é.) Por fim, o complemento de Schur é calculado pelas linhas 16-19 praticamente do mesmo modo que é calculado pelas linhas 7-12 de LU-Decomposition, exceto que aqui a operação é escrita para funcionar no lugar.

Devido à sua estrutura de laço triplamente aninhado, o tempo de execução de LUP-Decomposition é (n_3), que é igual ao de LU-Decomposition. Assim, pivotar nos custa no máximo um fator de tempo constante.

Exercícios

28.1-1 Resolva a equação

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

usando substituição direta.

Figura 28.2 A operação de LUP-DECOMPOSITION. (**a**) A matriz de entrada A com a permutação identidade das linhas à esquerda. A primeira etapa do algoritmo determina que o elemento 5 no círculo preto na terceira linha é o pivô para a primeira coluna. (**b**) As linhas 1 e 3 são trocadas e a permutação é atualizada. A coluna e a linha sombreadas representam v e w^T. (**c**) O vetor v é substituído por v / 5, e a parte inferior direita da matriz é atualizada com o complemento de Schur. Segmentos de reta dividem a matriz em três regiões: elementos de U (em cima), elementos de L (esquerda) e elementos do complemento de Schur (inferior direita). (**d**)-(**f**) A segunda etapa. (**g**)-(**i**) A terceira etapa. Nenhuma outra mudança ocorre na quarta e última etapa. (**j**) A decomposição LUP $PA = LU$.

28.1-2 Determine uma decomposição LU da matriz

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}$$

28.1-3 Resolva a equação

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

usando uma decomposição LUP.

28.1-4 Descreva a decomposição LUP de uma matriz diagonal.

28.1-5 Descreva a decomposição LUP de uma matriz de permutação A e prove que ela é única.

28.1-6 Mostre que, para todo $n \geq 1$, existe uma matriz singular $n \times n$ que tem uma decomposição LU.

28.1-7 Em LU-DECOMPOSITION, é necessário executar a iteração do laço **for** mais externo quando $k = n$? E em LUP-DECOMPOSITION?

28.2 INVERSÃO DE MATRIZES

Normalmente, não utilizamos inversas de matrizes para resolver sistemas de equações lineares na prática; em vez disso, preferimos usar técnicas numericamente mais estáveis como a decomposição LUP. Porém, às vezes, precisamos calcular a inversa de uma matriz. Nesta seção, mostramos como usar decomposição LUP para calcular a inversa de uma matriz. Provamos também que a multiplicação de matrizes e o cálculo da inversa de uma matriz são problemas de dificuldade equivalente, no sentido de que (sujeitos a condições técnicas) podemos usar um algoritmo escrito para um dos problemas para resolver o outro no mesmo tempo de execução assintótico. Assim, podemos usar o algoritmo de Strassen (veja Seção 4.2), escrito para multiplicação de matrizes, para inverter uma matriz. Na verdade, o artigo original publicado por Strassen foi motivado pelo problema de mostrar que um conjunto de equações lineares podia ser resolvido mais rapidamente do que pelo método usual.

Calculando a inversa de uma matriz a partir de uma decomposição LUP

Suponha que temos uma decomposição LUP de uma matriz A na forma de três matrizes L, U e P tais que $PA = LU$. Usando LUP-SOLVE, podemos resolver uma equação da forma $Ax = b$ no tempo (n_2). Visto que a decomposição LUP depende de A, mas não de b, podemos executar LUP-SOLVE para um segundo conjunto de equações da forma $Ax =$

b' no tempo adicional (n_2). Em geral, tão logo tenhamos a decomposição LUP de A, podemos resolver, no tempo (kn_2), k versões da equação $Ax = b$ cuja única diferença é o termo b.

Podemos considerar a equação

$$AX = I_n, \tag{28.10}$$

que define a matriz X, a inversa de A, como um conjunto de n equações distintas da forma $Ax = b$. Para sermos precisos, seja X_i a i-ésima coluna de X, e lembre-se de que o vetor unitário e_i é a i-ésima coluna de I_n. Então, podemos resolver a equação (28.10) para X, usando a decomposição LUP de A para resolver cada equação

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix} \text{ e } A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix}. \tag{28.11}$$

separadamente para X_i . Tão logo tenhamos a decomposição LUP, podemos calcular cada uma das n colunas X_i no tempo $(n\ _2)$ e, portanto, podemos calcular X pela decomposição LUP de A no tempo (n_3). Visto que podemos determinar a decomposição LUP de A no tempo (n_3), podemos calcular a inversa A-1 de uma matriz A no tempo (n_3).

Multiplicação de matrizes e inversão de matrizes

Agora, mostraremos que os fatores de aceleração teóricos obtidos para multiplicação de matrizes se traduzem em fatores de aceleração para inversão de matrizes. Na verdade, provamos algo mais forte: a inversão de matrizes é equivalente à multiplicação de matrizes no seguinte sentido: se $M(n)$ denota o tempo para multiplicar duas matrizes $n \times n$, então podemos inverter uma matriz $n \times n$ não singular no tempo $O(M(n))$. Ademais, se $I(n)$ denota o tempo para inverter uma matriz $n \times n$ não singular, podemos multiplicar duas matrizes $n \times n$ no tempo $O(I(n))$. Provamos esses resultados em dois teoremas separados.

Teorema 28.1 (Multiplicação não é mais difícil que inversão)

Se podemos inverter uma matriz $n \times n$ no tempo $I(n)$, onde $I(n) = (n_2)$ e $I(n)$ satisfazem a condição de regularidade $I(3n) = O(I(n))$, então podemos multiplicar duas matrizes $n \times n$ no tempo $O(I(n))$.

Prova Sejam A e B matrizes $n \times n$ cujo produto de matrizes C desejamos calcular. Definimos a matriz $3n \times 3n$ D por

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

A inversa de D é

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix},$$

e assim podemos calcular o produto AB tomando a submatriz superior direita $n \times n$ de D^{-1}. Podemos construir a matriz D no tempo (n_2), que é $O(I(n))$, porque supomos que $I(n) = (n_2)$, e podemos inverter D no tempo $O(I(3n)) = O(I(n))$, pela condição de regularidade para $I(n)$. Assim, temos $M(n) = O(I(n))$.

Observe que $I(n)$ satisfaz a condição de regularidade sempre que $I(n) = (n_c \lg^d n)$ para quaisquer constantes $c > 0$, $d \geq 0$.

A prova de que inversão de matrizes não é mais difícil que multiplicação de matrizes se baseia em algumas propriedades de matrizes simétricas positivas definidas que provaremos na Seção 28.3.

Teorema 28.2 (Inversão não é mais difícil que multiplicação)

Suponha que podemos multiplicar duas matrizes $n \times n$ reais no tempo $M(n)$, onde $M(n) = (n_2)$ e $M(n)$ satisfaz as duas condições de regularidade $M(n + k) = O(M(n))$ para qualquer k na faixa $0 \leq k \leq n$ e $M(n/2) \leq cM(n)$ para alguma constante $c < 1/2$. Então, podemos calcular a inversa de qualquer matriz $n \times n$ real não singular no tempo $O(M(n))$.

Prova Aqui provamos o teorema para matrizes reais. O Exercício 28.2-6 pede que você generalize a prova para matrizes cujas entradas são números complexos.

Podemos supor que n é uma potência exata de 2, já que temos

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}$$

para qualquer $k > 0$. Assim, escolhendo k tal que $n + k$ é uma potência de 2, aumentamos a matriz até um tamanho que é a próxima potência de 2 e obtemos a resposta desejada A^{-1} pela resposta ao problema aumentado. A primeira condição de regularidade para $M(n)$ assegura que esse aumento não provoca o aumento do tempo de execução por mais que um fator constante.

Por enquanto, vamos supor que a matriz $n \times n$ A seja simétrica e positiva definida. Particionamos A e sua inversa A^{-1} em quatro submatrizes $n/2 \times n/2$:

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix} \text{ e } A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix}. \tag{28.11}$$

Então, se fizermos de

$$S = D - CB^{-1}C^T \tag{28.12}$$

o complemento de Schur de A com relação a B (veremos mais detalhes sobre essa forma do complemento de Schur na Seção 28.3), temos

$$A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} = \begin{pmatrix} B^{-1} + B^{-1}C^T S^{-1} C B^{-1} & -B^{-1}C^T S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}, \tag{28.13}$$

já que $AA^{-1} = I$, como podemos verificar executando a multiplicação de matrizes. Como A é simétrica e positiva definida, os Lemas 28.4 e 28.5 na Seção 28.3 implicam que B e S são simétricas e positivas definidas. Portanto, pelo Lema 28.3 na Seção 28.3, as inversas B^{-1} e S^{-1} existem e, pelo Exercício D.2-6, B^{-1} e S^{-1} são simétricas, de modo que $(B^{-1})^T = B^{-1}$ e $(S^{-1})^T = S^{-1}$. Portanto, podemos calcular as submatrizes R, T, U e V de A^{-1} da maneira descrita a seguir, onde todas as matrizes mencionadas são $n/2 \times n/2$.

1. Forme as submatrizes B, C, C^T e D de A.
2. Calcule recursivamente a inversa B^{-1} de B.

3. Calcule o produto de matrizes $W = CB^{-1}$ e depois sua transposta W^T, que é igual a $B^{-1}C^T$ (pelo Exercício D.1-2 e $(B^{-1})^T = B^{-1}$).
4. Calcule o produto de matrizes $X = WC^T$, que é igual a $CB^{-1}C^T$, e depois a matriz $S = D - X = D - CB^{-1}C^T$.
5. Calcule recursivamente a inversa S^{-1} de S e defina V como S^{-1}.
6. Calcule o produto de matrizes $Y = S^{-1}W$, que é igual a $S^{-1}CB^{-1}$, e depois sua transposta Y^T, que é igual a $B^{-1}C^TS^{-1}$ (pelo Exercício D.1-2, $(B^{-1})^T = B^{-1}$ e $(S^{-1})^T = S^{-1}$). Defina T como $-Y^T$ e U como $-Y$.
7. Calcule o produto de matrizes $Z = W^TY$, que é igual a $B^{-1}C^TS^{-1}CB^{-1}$, e defina R como $B^{-1} + Z$.

Assim, podemos inverter uma matriz $n \times n$ simétrica positiva definida invertendo duas matrizes $n/2 \times n/2$ nas etapas 2 e 5, efetuando quatro multiplicações de matrizes $n/2 \times n/2$ nas etapas 3, 4, 6 e 7, mais um custo adicional de $O(n^2)$ para extrair submatrizes de A, inserir submatrizes em A^{-1} e executar um número constante de adições, subtrações e transpostas para matrizes $n/2 \times n/2$ matrizes. Obtemos a recorrência

$$\begin{aligned} I(n) &\le 2I(n/2) + 4M(n/2) + O(n^2) \\ &= 2I(n/2) + \Theta(M(n)) \\ &= O(M(n)). \end{aligned}$$

A segunda linha é válida porque a condição de regularidade no enunciado do teorema implica que $4M(n/2) < 2M(n)$ e porque consideramos que $M(n) = (n^2)$. A terceira linha decorre porque a segunda condição de regularidade nos permite aplicar o caso 3 do teorema mestre (Teorema 4.1).

Resta provar que podemos obter o mesmo tempo de execução assintótico para multiplicação de matrizes que o obtido para inversão de matrizes quando A é inversível mas não simétrica e positiva definida. A ideia básica é que, para qualquer matriz não singular A, a matriz A^TA é simétrica (pelo Exercício D.1-2) e positiva definida (pelo Teorema D.6). Então, o artifício é reduzir o problema de inverter A ao problema de inverter A^TA.

A redução é baseada na seguinte observação: quando A é uma matriz $n \times n$ não singular, temos

$$A^{-1} = (A^TA)^{-1}A^T ,$$

já que $((A^TA)^{-1}A^T)A = (A^TA)^{-1}(A^TA) = I$ e uma matriz inversa é única. Portanto, podemos calcular A^{-1} primeiro multiplicando A^T por A para obter A^TA e depois invertendo a matriz simétrica positiva definida A^TA empregando o algoritmo de divisão e conquista, e finalmente multiplicando o resultado por A^T. Cada uma dessas três etapas demora o tempo $O(M(n))$ e, assim, podemos inverter qualquer matriz não singular com entradas reais no tempo $O(M(n))$.

A prova do Teorema 28.2 sugere um meio de resolver a equação $Ax = b$ usando decomposição LU sem pivotação, desde que A seja não singular. Multiplicamos ambos os lados da equação por A^T, o que produz $(A^TA)x = A^Tb$. Essa transformação não afeta a solução x, já que A^T é inversível, e assim podemos fatorar a matriz simétrica positiva definida A^TA calculando uma decomposição LU. Então, usamos substituição direta e inversa para resolver para x com o lado direito A^Tb. Embora esse método esteja teoricamente correto, na prática o procedimento LUP-DECOMPOSITION funciona muito melhor. A decomposição LUP requer menor número de operações aritméticas por um fator constante e tem propriedades numéricas um pouco melhores.

Exercícios

28.2-1 Seja $M(n)$ o tempo para multiplicar duas matrizes $n \times n$ e seja $S(n)$ o tempo necessário para elevar uma matriz $n \times n$ ao quadrado. Mostre que multiplicar e elevar matrizes ao quadrado têm essencialmente a mesma dificuldade: um algoritmo de multiplicação de matrizes de tempo $M(n)$ implica um algoritmo de elevação ao quadrado de tempo $O(M(n))$, e um algoritmo de elevação ao quadrado de tempo $S(n)$ implica um algoritmo de multiplicação de matrizes de tempo $O(S(n))$.

28.2-2 Seja $M(n)$ o tempo para multiplicar duas matrizes $n \times n$ e seja $L(n)$ o tempo para calcular a decomposição LUP de uma matriz $n \times n$. Mostre que um algoritmo de multiplicação de matrizes de tempo $M(n)$ implica um algoritmo de decomposição LUP de tempo $O(M(n))$.

28.2-3 Seja $M(n)$ o tempo para multiplicar duas matrizes $n \times n$ e seja $D(n)$ o tempo necessário para encontrar o determinante de uma matriz $n \times n$. Mostre que multiplicar matrizes e calcular o determinante têm essencialmente a mesma dificuldade: um algoritmo de multiplicação de matrizes de tempo $M(n)$ implica um algoritmo de determinante de tempo $O(M(n))$, e um algoritmo de determinante de tempo $D(n)$ implica um algoritmo de multiplicação de matrizes de tempo $O(D(n))$.

28.2-4 Seja $M(n)$ o tempo para multiplicar duas matrizes booleanas $n \times n$ e seja $T(n)$ o tempo para determinar o fecho transitivo de uma matriz booleana $n \times n$ (veja a Seção 25.2). Mostre que um algoritmo de multiplicação de matrizes booleanas de tempo $M(n)$ implica um algoritmo de fecho transitivo de tempo $O(M(n) \lg n)$, e um algoritmo de fecho transitivo de tempo $T(n)$ implica um algoritmo de multiplicação de matrizes booleanas de tempo $O(T(n))$.

28.2-5 O algoritmo de inversão de matrizes baseado no Teorema 28.2 funciona quando elementos de matrizes são retirados do corpo de inteiros módulo 2? Explique.

28.2-6 ★ Generalize o algoritmo de inversão de matrizes do Teorema 28.2 para tratar matrizes de números complexos e prove que sua generalização funciona corretamente. (*Sugestão*: Em vez da transposta de A, use a ***transposta conjugada*** A_*, que é obtida da transposta de A pela substituição de cada entrada por seu conjugado complexo. Em vez de matrizes simétricas, considere matrizes ***hermitianas***, que são matrizes A tais que $A = A_*$.)

28.3 Matrizes simétricas positivas definidas e aproximação de mínimos quadrados

Matrizes simétricas positivas definidas têm muitas propriedades interessantes e desejáveis. Por exemplo, elas são não singulares, e podemos executar decomposição LU com elas sem precisar nos preocupar com divisão por zero. Nesta seção, provaremos várias outras propriedades importantes de matrizes simétricas positivas definidas e mostraremos uma aplicação interessante para ajuste de curvas por uma aproximação de mínimos quadrados.

A primeira propriedade que provamos talvez seja a mais básica.

Lema 28.3

Qualquer matriz simétrica positiva definida é não singular.

Prova Suponha que uma matriz A seja singular. Então, pelo Corolário D.3 existe um vetor não nulo x tal que $Ax = 0$. Por consequência, $x_T Ax = 0$ e A não pode ser positiva definida.

A prova de que podemos executar a decomposição LU para uma matriz simétrica positiva definida A sem dividir por zero é mais complicada. Começamos provando propriedades de certas submatrizes de A. Defina a k-ésima ***submatriz líder*** de A como a matriz A_k que consiste na interseção das primeiras k linhas e das primeiras k colunas de A.

Lema 28.4

Se A é uma matriz simétrica positiva definida, então toda submatriz líder de A é simétrica e positiva definida.

Prova É óbvio que cada submatriz líder A_k é simétrica. Para provar que A_k é positiva definida, suponhamos que ela não seja e deduzimos uma contradição. Se A_k não é positiva definida, então existe um k-vetor $x_k \neq 0$ tal que $x_T{}^k A_k x_k \leq 0$. Tomando A como $n \times n$ e

$$A = \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \tag{28.14}$$

para submatrizes B (que é $(n - k)\times k$) e C (que é $(n - k)\times(n - k)$). Defina o n-vetor $x = (x_T{}^k\ 0)^T$, onde $n - k$ zeros vêm depois de x_T. Então temos

$$\begin{aligned} x^T A x &= (x_k^T \quad 0)\begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}\begin{pmatrix} x_k \\ 0 \end{pmatrix} \\ &= (x_k^T \quad 0)\begin{pmatrix} A_k x_k \\ B x_k \end{pmatrix} \\ &= x_k^T A_k x_k \\ &\leq 0, \end{aligned}$$

o que contradiz a afirmação de que A seja positiva definida.

Agora, voltamos a algumas propriedades essenciais do complemento de Schur. Seja A uma matriz simétrica positiva definida e seja A_k uma submatriz líder $k \times k$ de A. Particione A mais uma vez de acordo com a equação (28.14). Generalizamos a equação (28.9) para definir o ***complemento de Schur*** S de A com relação a A_k como

$$S = C - BA_k^{-1}B^T. \tag{28.15}$$

(Pelo Lema 28.4, A_k é simétrica e positiva definida; portanto, A^k existe pelo Lema 28.3, e S é bem definida.) Observe que nossa primeira definição (28.9) do complemento de Schur é consistente com a equação (28.15) se fizermos $k = 1$.

O próximo lema mostra que as matrizes de complemento de Schur de matrizes simétricas positivas definidas são também simétricas e positivas definidas. Usamos esse resultado no Teorema 28.2, e precisamos de seu corolário para provar a correção da decomposição LU para matrizes simétricas positivas definidas.

Lema 28.5 (Lema do complemento de Schur)

Se A é uma matriz simétrica positiva definida e A_k é uma submatriz líder $k \times k$ de A, então o complemento de Schur de A com relação a A_k é simétrico e positivo definido.

Prova Como A é simétrica, a submatriz C também é. Pelo Exercício D.2-6, o produto $BA_k{-1}B_T$ é simétrico e, pelo Exercício D.1-1, S é simétrica.

Resta mostrar que S é positiva definida. Considere a partição de A dada na equação (28.14). Para qualquer vetor não nulo x, temos $x_T Ax > 0$ pela hipótese de A ser positiva definida. Vamos decompor x em dois subvetores y e z compatíveis com A_k e C, respectivamente. Como existe, temos

$$
\begin{aligned}
x^{\mathrm{T}}Ax &= (y^{\mathrm{T}} \quad z^{\mathrm{T}})\begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix} \\
&= (y^{\mathrm{T}} \quad z^{\mathrm{T}})\begin{pmatrix} A_k y + B^{\mathrm{T}} z \\ By + Cz \end{pmatrix} \\
&= y^{\mathrm{T}}A_k y + y^{\mathrm{T}}B^{\mathrm{T}}z + z^{\mathrm{T}}By + z^{\mathrm{T}}Cz \\
&= (y + A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}}\, A_k(y + A_k^{-1}\, B^{\mathrm{T}}z) + z^{\mathrm{T}}\,(C - BA_k^{-1}\, B^{\mathrm{T}})z\,, \qquad (28.16)
\end{aligned}
$$

por mágica de matrizes (verifique efetuando multiplicações). Esta última equação equivale a "completar o quadrado" da forma quadrática (veja o Exercício 28.3-2).

Visto que $x_{\mathrm{T}}Ax > 0$ é válida para qualquer x não nulo, qualquer z não nulo para podermos escolher $y = - A^{-1}k\ B_{\mathrm{T}}z$, o que faz o primeiro termo da equação (28.16) desaparecer, restando

$$
z^{\mathrm{T}}(C - BA_k^{-1}\, B^{\mathrm{T}})z = z^{\mathrm{T}}Sz
$$

como o valor da expressão. Portanto, para qualquer $z \neq 0$, temos $z_{\mathrm{T}}Sz = x_{\mathrm{T}}Ax > 0$ e, assim, S é positiva definida.

Corolário 28.6

A decomposição LU de uma matriz simétrica positiva definida nunca provoca uma divisão por zero.

Prova Seja A uma matriz simétrica positiva definida. Provaremos algo mais forte que o enunciado do corolário: todo pivô é estritamente positivo. O primeiro pivô é a_{11}. Seja e_1 o primeiro vetor unitário, do qual obtemos $a_{11} = e^{\mathrm{T}1}\, Ae_1 > 0$. Visto que a primeira etapa da decomposição LU produz o complemento de Schur de A com relação a $A_1 = (a_{11})$, o Lema 28.5 implica por indução que todos os pivôs são positivos.

Aproximação de mínimos quadrados

Uma aplicação importante de matrizes simétricas positivas definidas é o ajuste de curvas a determinados conjuntos de pontos de dados. Suponha que tenhamos um conjunto de m pontos de dados

$$
(x_1, y_1), (x_2, y_2), \ldots, (x_{m,} y_m)\ ,
$$

onde sabemos que os valores y_{i} estão sujeitos a erros de medição. Gostaríamos de determinar uma função $F(x)$ tal que os erros de aproximação

$$
\eta_i = F(x^i) - y_i \qquad (28.17)
$$

são pequenos para $i = 1, 2, \ldots, m$. A forma da função F depende do problema em questão. Aqui consideramos a forma de uma soma linearmente ponderada,

$$
F(x) = \sum_{j=1}^{n} c_j f_j(x),
$$

onde o número de somandos n e as ***funções de base*** f_{j} específicas são escolhidos com base no conhecimento do problema em questão. Uma opção comum é $f_{\mathrm{j}}(x) = x_{\mathrm{j}}\ -\ ^1$, o que significa que

$$
F(x) = c_1 + c_2x + c_3x^2 + + c_nx^{n-1}
$$

é um polinômio de grau n - 1 em x. Assim, dados m pontos de dados (x_1, y_1), (x_2, y_2), …, (x_m, y_m), desejamos calcular n coeficientes $c_1, c_2, \ldots, c_n$ que minimizem os erros de aproximação $_1$, $_2$, …, $_m$.

Escolhendo $n = m$, podemos calcular cada y_i *exatamente* na equação (28.17). Porém, tal função F de grau elevado "ajusta o ruído", bem como os dados, e em geral produz resultados ruins quando usada com a finalidade de predizer y para valores de x que ainda não foram vistos. Normalmente é melhor escolher n significativamente menor que m e esperar que, escolhendo bem os coeficientes c_j, possamos obter uma função F que determine os padrões significativos nos pontos de dados sem prestar atenção indevida ao ruído. Existem alguns princípios teóricos para escolher n, mas eles não fazem parte do escopo deste texto. Em todo caso, uma vez escolhido um valor de n menor que m, acabamos com um conjunto superdeterminado de equações cuja solução desejamos aproximar. Mostramos agora como fazer isso.

Seja

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix}$$

a matriz de valores das funções básicas nos pontos dados; isto é, $a_{ij} = f_j(x_i)$. Seja $c = (c_k)$ o desejado vetor n de coeficientes. Então,

$$\begin{aligned} Ac &= \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \\ &= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix} \end{aligned}$$

é o vetor m de "valores previstos" para y. Assim,

$$\eta = Ac - y$$

é o vetor m de ***erros de aproximação***.

Para minimizar erros de aproximação, optamos por minimizar a norma do vetor de erro , o que nos dá uma ***solução de mínimos quadrados***, já que

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2 \right)^{1/2} .$$

Como

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right)^2 ,$$

podemos minimizar η diferenciando η^2 em relação a cada c_k e, então, exigindo que o resultado seja 0:

$$\frac{d\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2 \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right) a_{ik} = 0. \tag{28.18}$$

As n equações (28.18) para $k = 1, 2, ..., n$ são equivalentes à única equação matricial $(Ac - y)^T A = 0$

$$(Ac - y)^T A = 0$$

ou, o que é equivalente (usando o Exercício D.1-2), a

$$A^T (Ac - y) = 0$$

que implica

$$A^T Ac = A^T y. \tag{28.19}$$

Em estatística essa expressão é denominada ***equação normal***. A matriz $A_T A$ é simétrica pelo Exercício D.1-2 e, se A tem posto total de colunas, pelo Teorema D.6 $A_T A$ também é positiva definida. Por consequência, $(A_T A)^{-1}$ existe, e a solução para a equação (28.19) é

$$\begin{aligned} c &= ((A^T A)^{-1} A^T) y \\ &= A^+ y , \end{aligned} \tag{28.20}$$

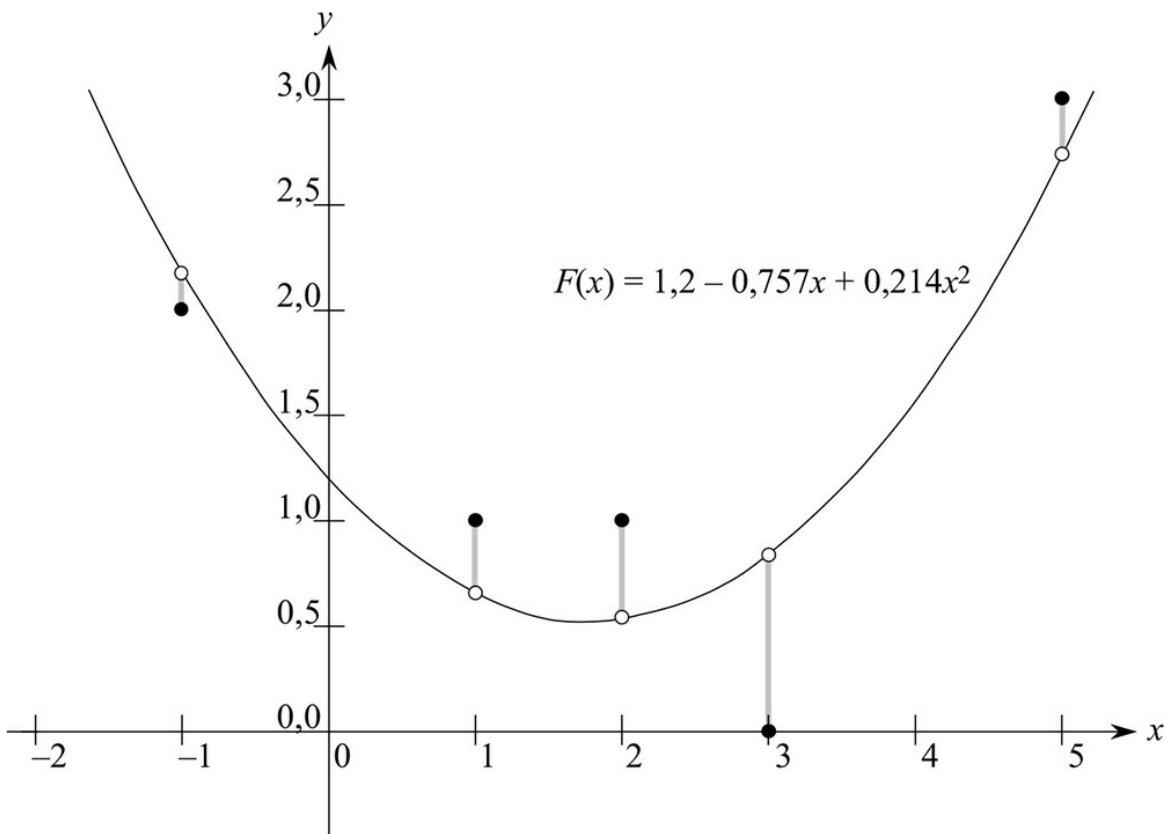
onde a matriz $A+ = ((A_T A)^{-1} A_T)$ é denominada ***pseudoinversa*** da matriz A. A pseudoinversa generaliza naturalmente a noção de uma matriz inversa para o caso em que A não é quadrada. (Compare a equação (28.20) como a solução aproximada para $Ac = y$ com a solução $A^{-1}b$ como a solução exata para $Ax = b$.)

Como exemplo da produção de um ajuste de mínimos quadrados, suponha que tenhamos cinco pontos de dados

$$(x_1, y_1) = (-1, 2) ,$$

$$(x_2, y_2) = (1, 1) ,$$

$$(x_3, y_3) = (2, 1) ,$$

$$(x_4, y_4) = (3, 0) ,$$

$$(x_5, y_5) = (5, 3) ,$$

mostrados como pontos pretos na Figura 28.3. Desejamos ajustar esses pontos com um polinômio quadrático

$$F(x) = c_1 + c_2 x + c_3 x^2 \, .$$

Começamos com a matriz de valores de funções de base

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix} .$$

Figura 28.3 O ajuste de mínimos quadrados de um polinômio quadrático ao conjunto de cinco pontos de dados {(1, 2), (1, 1), (2, 1), (3, 0), (5, 3)}. Os pontos pretos são os pontos de dados, e os pontos brancos são seus valores estimados previstos pelo polinômio $F(x) = 1{,}2 - 0{,}757x + 0{,}214x_2$, o polinômio quadrático que minimiza a soma dos erros elevados ao quadrado. Cada linha sombreada mostra o erro para um ponto de dado.

cuja pseudoinversa é

$$A^+ = \begin{pmatrix} 0{,}500 & 0{,}300 & 0{,}200 & 0{,}100 & -0{,}100 \\ -0{,}388 & 0{,}093 & 0{,}190 & 0{,}193 & -0{,}088 \\ 0{,}060 & -0{,}036 & -0{,}048 & -0{,}036 & 0{,}060 \end{pmatrix}.$$

Multiplicando y por $A+$, obtemos o vetor de coeficientes

$$c = \begin{pmatrix} 1{,}200 \\ -0{,}757 \\ 0{,}214 \end{pmatrix},$$

que corresponde ao polinômio quadrático

$$F(x) = 1{,}200 - 0{,}757x + 0{,}214x^2$$

como o polinômio que melhor se ajusta aos dados especificados, em um sentido de mínimos quadrados.

Por uma questão prática, resolvemos a equação normal (28.19) multiplicando y por A_T e determinando uma decomposição LU de A_TA. Se A tem posto total, é garantido que a matriz A_TA é não singular porque é simétrica e positiva definida (Veja o Exercício D.1-2 e o Teorema D.6).

Exercícios

28.3-1 Prove que todo elemento da diagonal de uma matriz simétrica positiva definida é positivo.

28.3-2 Seja $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ uma matriz 2 × 2 simétrica positiva definida. Prove que seu determinante ac - b_2 é positivo "completando o quadrado" de maneira semelhante à usada na prova do Lema 28.5.

28.3-3 Prove que o elemento máximo em uma matriz simétrica positiva definida encontra-se na diagonal.

28.3-4 Prove que o determinante de cada submatriz líder de uma matriz simétrica positiva definida é positivo.

28.3-5 Seja A_k a k-ésima submatriz líder de uma matriz simétrica positiva definida A. Prove que det(A_k)/det(A_k-1) é o k-ésimo pivô durante a decomposição LU onde, por convenção, det(A_0) = 1.

28.3-6 Determine a função da forma

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

que é o melhor ajuste de mínimos quadrados para os pontos de dados

$$(1, 1), (2, 1), (3, 3), (4, 8) .$$

28.3-7 Mostre que a pseudoinversa $A+$ satisfaz as quatro equações a seguir:

$$
\begin{aligned}
AA^+ A &= A \,, \\
A^+ AA^+ &= A^+ \,, \\
(AA^+)^{\mathrm{T}} &= AA^+ \,, \\
(A^+A)^{\mathrm{T}} &= A^+ A \,.
\end{aligned}
$$

Problemas

28-1 Sistemas tridiagonais de equações lineares

Considere a matriz tridiagonal

$$
A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.
$$

a. Determine uma decomposição LU de A.

b. Resolva a equação $Ax = (1\ 1\ 1\ 1\ 1)^{\mathrm{T}}$ usando substituição direta e inversa.

c. Determine a inversa de A.

d. Mostre, para qualquer matriz tridiagonal $n \times n$ simétrica positiva definida A e qualquer vetor b de n elementos, como resolver a equação $Ax = b$ no tempo $O(n)$, executando uma decomposição LU. Demonstre que qualquer método baseado na formação de A^{-1} é assintoticamente mais caro no pior caso.

e. Mostre, para qualquer matriz tridiagonal $n \times n$ não singular A e qualquer vetor b de n elementos, como resolver a equação $Ax = b$ no tempo $O(n)$, executando uma decomposição LUP.

28-2 Splines

Um método prático para interpolar um conjunto de pontos com uma curva é usar ***spli- nes cúbicos.*** Temos um conjunto $\{(x_i, y_i) : i = 0, 1, ..., n\}$ de $n + 1$ pares de valores de pontos, onde $x_0 < x_1 < < x_n$. Desejamos ajustar uma curva cúbica por partes (spline) $f(x)$ aos pontos. Isto é, a curva $f(x)$ é formada por n polinômios cúbicos $f_i(x) = a_i + b_{ix} + c_{ix}2 + d_{ix}3$ para $i = 0, 1, ..., n - 1$ onde se x cair na faixa $x\ {}^{i}\ x\ x\ {}_{i} + {}^{1}$, o valor da curva será dado por $f(x) = f_i(x - x_i)$. Os pontos x_i onde os polinômios cúbicos são "colados" um ao outro são denominados ***nós***. Por simplicidade, suporemos que $x_i = i$ para $i = 0, 1, ..., n$.

Para garantir continuidade de $f(x)$, é necessário que

$$
\begin{aligned}
f(x_i) &= f_i(0) = y_i\ ,\\
f(x_{i+1}) &= f_i(1) = y_{i+1}
\end{aligned}
$$

para $i = 0, 1, ..., n - 1$. Para garantir que $f(x)$ seja suficientemente suave, também insistimos em que a derivada de primeiro grau seja contínua em cada nó:

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0)$$

para $i = 0, 1, ..., n - 1$.

a. Suponha que, para $i = 0, 1, ... n$, tenhamos não somente os pares de valores de pontos $\{(x_i, y_i)\}$, mas também as derivadas $D_i = f\ '(x_i)$ em cada nó. Expresse cada coeficiente a_i, b_i, c_i e d_i em termos dos valores y_i, y_{i+1}, D_i e D_{i+1}. (Lembre-se de que $x_i = i$.) Com que rapidez podemos calcular os $4n$ coeficientes pelos pares de valores de pontos e derivadas de primeiro grau?

Ainda resta a questão de como escolher as derivadas de $f(x)$ nos nós. Um método é exigir que as derivadas segundas sejam contínuas nos nós:

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

para $i = 0, 1, ..., n - 1$. No primeiro e no último nó, consideramos que $f''(x_{0}) = f''_{0}(0) = 0$ e $f''(x_n) = f''_{n-1}(1) = 0$; essas hipóteses fazem de $f(x)$ um spline cúbico ***natural***.

b. Use as restrições de continuidade na derivada segunda para mostrar que, para $i = 1, 2, ... n - 1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) \qquad (28.21)$$

c. Mostre que

$$2D_0 + D_1 = 3(y_1 - y_0)\ , \qquad (28.22)$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1})\ . \qquad (28.23)$$

d. Reescreva as equações (28.21) a (28.23) como uma equação matricial envolvendo o vetor $D = D_0, D_1, ..., D_n$ de incógnitas. Quais atributos têm a matriz em sua equação?

e. Demonstre que uma spline cúbica natural pode interpolar um conjunto de $n + 1$ pares de valores de pontos no tempo $O(n)$ (veja o Problema 28-1).

f. Mostre como determinar um spline cúbico natural que interpola um conjunto de $n + 1$ pontos (x_i, y_i) satisfazendo $x_0 < x_1 < ... < x_n$, mesmo quando x_i não é necessariamente igual a i. Qual equação matricial seu método deve resolver e com que rapidez seu algoritmo é executado?

NOTAS DO CAPÍTULO

Muitos textos excelentes descrevem computação numérica e científica com muito mais detalhes do que o espaço nos permite aqui. Os textos a seguir são especialmente interessantes: George e Liu [132], Golub e Van Loan [144], Press, Teukolsky, Vetterling e Flannery [283, 284] e Strang [323, 324].

Golub e Van Loan [144] discutem estabilidade numérica. Eles mostram por que $\det(A)$ não é necessariamente um bom indicador da estabilidade de uma matriz A, propondo em vez disso, usar $\|A\|_\infty \|A^{-1}\|_\infty$, onde $\|A\|_\infty = \max_{l \le i \le n} \sum_{j=1}^{n} |a_{ij}|$. Eles também abordam a questão de como calcular esse valor sem calcular A^{-1}.

O método de eliminação de Gauss, no qual se baseiam as decomposições LU e LUP, foi o primeiro método sistemático para resolver sistemas de equações lineares. Também foi um dos primeiros algoritmos numéricos. Embora fosse conhecido antes, sua descoberta é comumente atribuída a C. F. Gauss (1777-1855). Em seu famoso artigo [325], Strassen mostrou que uma matriz $n \times n$ pode ser invertida no tempo $O(n^{\lg 7})$. Winograd [358] provou originalmente que multiplicação de matrizes não é mais difícil que inversão de matrizes, e a recíproca se deve a Aho, Hopcroft e Ullman [5].

Outra decomposição de matrizes importante é a ***decomposição de valor singular***, ou ***SVD*** (*singular value decomposition*). A SVD fatora uma matriz $m \times n$ A em $A = Q_1 \sum Q_2^T$, onde é uma matriz $m \times n$ com valores não nulos somente na diagonal, Q_1 é $m \times m$ com colunas mutuamente ortonormais e Q_2 é $n \times n$, também com colunas mutuamente ortonormais. Dois vetores são ***ortonormais*** se seu produto interno é 0 e cada vetor tem uma norma de 1. Os livros de Strang [323, 324] e de Golub e Van Loan [144] contêm bons tratamentos da SVD.

Strang [324] tem uma apresentação excelente de matrizes simétricas positivas definidas e de álgebra linear em geral.

29 PROGRAMAÇÃO LINEAR

Muitos problemas consistem em maximizar ou minimizar um objetivo, dados recursos limitados e restrições concorrentes. Se pudermos especificar o objetivo como uma função linear de certas variáveis e se pudermos especificar as restrições aos recursos como igualdades ou desigualdades lineares nessas variáveis, teremos um ***problema de programação linear***. Programas lineares surgem em uma variedade de aplicações práticas. Começamos estudando uma aplicação em política eleitoral.

Um problema político

Suponha que você seja um político tentando vencer uma eleição. Seu eleitorado está distribuído por três tipos diferentes de áreas: urbana, suburbana e rural. Essas áreas têm, respectivamente, 100.000, 200.000 e 50.000 eleitores registrados. Embora nem todos os eleitores registrados se apresentem para votar, você decide que para governar efetivamente gostaria que, no mínimo, metade dos eleitores registrados em cada uma das três regiões vote em você. Você é honrado e nunca consideraria apoiar políticas nas quais não acredita, mas percebe que certas questões têm mais poder para conquistar votos em certos lugares. As questões primordiais de sua agenda de candidato são construção de mais estradas, controle de armas, subsídios agrícolas e um imposto sobre combustíveis destinado à melhoria do trânsito. De acordo com as pesquisas da sua equipe de campanha, você pode estimar quantos votos conquistará ou perderá em cada segmento da população gastando \$1.000 em propaganda para cada questão. Essa informação aparece na tabela da Figura 29.1. Nessa tabela, cada entrada descreve o número de milhares de eleitores urbanos, suburbanos ou rurais que você conquistaria se gastasse \$1.000 com propaganda em defesa de uma questão específica. Entradas negativas denotam votos que seriam perdidos. Sua tarefa é determinar a quantidade mínima de dinheiro que seria preciso gastar para obter 50.000 votos urbanos, 100.000 votos suburbanos e 25.000 votos rurais.

Se bem que você poderia criar por tentativa e erro uma estratégia que conquistasse o número necessário de votos, tal estratégia poderia não ser a menos dispendiosa. Por exemplo, você poderia dedicar \$20.000 à propaganda para construção de estradas, \$0 para controle de armas, \$4.000 para subsídios agrícolas e \$9.000 para um imposto sobre combustíveis. Nesse caso, você conquistaria $20(-2) + 0(8) + 4(0) + 9(10) = 50$ mil votos urbanos, $20(5) + 0(2) + 4(0) + 9(0) = 100$ mil votos suburbanos e $20(3) + 0(-5) + 4(10) + 9(-2) = 82$ mil votos rurais. Essas seriam as quantidades exatas de votos desejados nas áreas urbanas e suburbanas, mais que a quantidade de votos suficientes na área rural. (Na verdade, na área rural você receberia mais votos que o número de eleitores existentes!) Para arrebanhar esses votos, você teria pago $20 + 0 + 4 + 9 = 33$ mil dólares em propaganda.

política	urbanos	suburbanos	rurais
construir estradas	−2	5	3
controle de armas	8	2	−5
subsídios agrícolas	0	0	10
imposto sobre combustíveis	10	0	−2

Figura 29.1 O efeito das políticas sobre os eleitores. Cada entrada descreve o número de milhares de eleitores urbanos, suburbanos ou rurais que poderiam ser conquistados gastando $1.000 em propaganda de apoio a uma política para uma questão específica. Entradas negativas denotam votos que seriam perdidos.

Seria natural você pensar se essa estratégia é a melhor possível. Isto é, será que você poderia alcançar suas metas gastando menos com propaganda? Um processo adicional de tentativa e erro poderia ajudá-lo a responder a essa pergunta, mas não seria ótimo ter um método sistemático para responder a tais perguntas? Para desenvolver tal método formulamos essa questão em termos matemáticos. Introduzimos quatro variáveis:

- x_1 é o número de milhares de dólares gastos com a propaganda da construção de estradas,
- x_2 é o número de milhares de dólares gastos com a propaganda do controle de armas,
- x_3 é o número de milhares de dólares gastos com a propaganda dos subsídios agrícolas,
- x_4 é o número de milhares de dólares gastos com a propaganda do imposto sobre combustíveis.

O requisito para conquistar no mínimo 50.000 votos urbanos pode ser expresso como

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \,. \qquad (29.1)$$

De modo semelhante, podemos expressar os requisitos para conquistar no mínimo 100.000 votos suburbanos e 25.000 votos rurais como

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \qquad (29.2)$$

e

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \,. \qquad (29.3)$$

Qualquer configuração das variáveis x_1, x_2, x_3, x_4 que satisfaça as desigualdades (29.1)- (29.3) produz uma estratégia que conquista um número suficiente de cada tipo de voto. Para manter os custos tão baixos quanto possível, você gostaria de minimizar a quantia gasta com propaganda. Isto é, o que você quer é minimizar a expressão

$$x_1 + x_2 + x_3 + x_4 \,. \qquad (29.4)$$

Embora seja comum ocorrer ***propaganda negativa*** em campanhas políticas, certamente não existe ***propaganda de custo negativo***. Consequentemente, temos de impor a seguinte condição:

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0 \text{ e } x_4 \geq 0 \,. \qquad (29.5)$$

Combinando as desigualdades (29.1)-(29.3) e (29.5) com o objetivo de minimizar (29.4), obtemos o que é conhecido como “programa linear”. Formatamos esse problema como

$$
\begin{aligned}
&\text{minimizar} \quad x_1 + x_2 + x_3 + x_4. && (29.6)\\
&\text{sujeito a}\\
&\quad -2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 && (29.7)\\
&\quad 5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 && (29.8)\\
&\quad 3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 && (29.9)\\
&\quad x_1, x_2, x_3, x_4 \geq 0. && (29.10)
\end{aligned}
$$

A solução desse programa linear produz sua estratégia ótima.

Programas lineares gerais

No problema geral de programação linear, desejamos otimizar uma função linear sujeita a um conjunto de desigualdades lineares. Dado um conjunto de números reais a_1, a_2, ..., a_n e um conjunto de variáveis x_1, x_2, ..., x_n, definimos uma ***função linear*** f dessas variáveis por

$$f(x_1, x_2, ..., x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = f(x_1, \cdots, x_n) = a_1x_2 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^{n} a_jx_j \cdot$$

Se b é um número real e f é uma função linear, então a equação

$$f(x_1, x_2, ..., x_n) = b$$

é uma ***igualdade linear*** e as desigualdades

$$f(x_1, x_2, ..., x_n) \leq b$$

e

$$f(x_1, x_2, ..., x_n) \geq b$$

são ***desigualdades lineares***. Usamos a expressão geral ***restrições lineares*** para denotar igualdades lineares ou desigualdades lineares. Em programação linear, não permitimos desigualdades estritas. Formalmente, um ***problema de programação linear*** é o problema de minimizar ou maximizar uma função linear sujeita a um conjunto finito de restrições lineares. Se formos minimizar, denominamos o programa linear ***programa linear de minimização*** e, se formos maximizar, denominamos o programa linear ***programa linear de maximização***.

O restante deste capítulo abrangerá a formulação e a solução de programas lineares. Embora vários algoritmos de tempo polinomial para programação linear tenham sido desenvolvidos, não os estudaremos neste capítulo. Em vez disso, estudaremos o algoritmo simplex, que é o mais antigo algoritmo de programação linear. O algoritmo simplex não é executado em tempo polinomial no pior caso, mas é razoavelmente eficiente e muito utilizado na prática.

Uma visão geral da programação linear

Para descrever propriedades e algoritmos de programas lineares, é conveniente expressá-los em formas canônicas. Neste capítulo utilizaremos duas formas: a ***forma-padrão*** e a ***forma de folgas.*** Daremos as definições exatas na Seção 29.1. Informalmente, um programa linear na forma-padrão é a maximização de uma função linear sujeita a *desigualdades* lineares, enquanto um programa linear em forma de folgas é a maximização de uma função linear sujeita a *igualdades* lineares. Em geral, utilizaremos a forma-padrão para expressar programas lineares, mas achamos mais conveniente usar a forma de folgas quando descrevemos os detalhes do algoritmo simplex. Por enquanto, restringimos nossa atenção a maximizar uma função linear de n variáveis sujeita a um conjunto de m desigualdades lineares.

Vamos considerar primeiro o seguinte programa linear com duas variáveis:

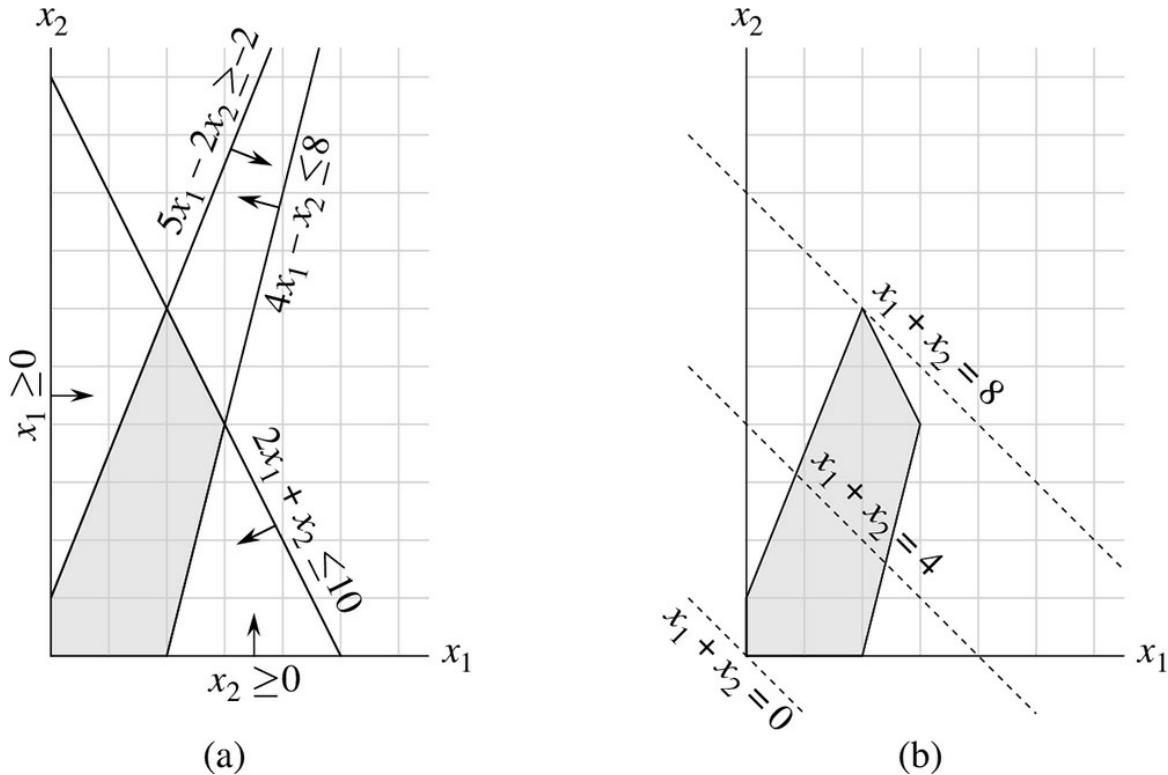
$$
\begin{array}{lrcrcrr}
\text{maximizar} & x_1 & + & x_2 & & & (29.11) \\
\text{sujeito a} & & & & & & \\
& 4x_1 & - & x_2 & \leq & 8 & (29.12) \\
& 2x_1 & + & x_2 & \leq & 10 & (29.13) \\
& 5x_1 & - & 2x_2 & \geq & -2 & (29.14) \\
& & x_1, x_2 & & \leq & 0 & (29.15)
\end{array}
$$

Denominamos qualquer configuração das variáveis x_1 e x_2 que satisfaça todas as restrições (29.12)-(29.15) uma ***solução viável*** para o programa linear. Se representarmos as restrições em um gráfico de coordenadas cartesianas (x_1 , x_2), como na Figura 29.2(a), vemos que o conjunto de soluções viáveis (sombreado na figura) forma uma região convexa[1] no espaço bidimensional. Denominamos essa região convexa ***região viável*** e denominamos ***função objetivo*** a função que desejamos maximizar. Conceitualmente, poderíamos avaliar a função objetivo $x_1 + x_2$ em cada ponto da região viável; damos o nome de ***valor objetivo*** ao valor da função objetivo em um determinado ponto. Então, podemos identificar um ponto que tem o valor objetivo máximo como uma solução ótima. Para esse exemplo (e para a maioria dos programas lineares), a região viável contém um número infinito de pontos e, portanto, precisamos determinar um modo eficiente de encontrar um ponto que alcance o valor objetivo máximo sem avaliar explicitamente a função objetivo em cada ponto na região viável.

Figura 29.2 (**a**) O programa linear dado em (29.12)-(29.15). Cada restrição é representada por uma reta e uma direção. A interseção das restrições, que é a região viável, está sombreada. (**b**) As retas pontilhadas mostram, respectivamente, os pontos para os quais o valor objetivo é 0, 4 e 8. A solução ótima para o programa linear é $x_1 = 2$ e $x_2 = 6$ com valor objetivo 8.

Em duas dimensões, podemos otimizar por meio de um procedimento gráfico. O conjunto de pontos para os quais $x_1 + x_2 = z$, para qualquer z, é uma reta de inclinação -1. Se representarmos $x_1 + x_2 = 0$ em gráfico, obteremos a reta de inclinação -1 que passa pela origem, como na Figura 29.2(b). A interseção dessa reta com a região viável é o conjunto de soluções viáveis que têm um valor objetivo 0. Nesse caso, essa interseção da reta com a região viável é o ponto único (0, 0).

De modo mais geral, para qualquer z, a interseção da reta $x_1 + x_2 = z$ com a região viável é o conjunto de soluções viáveis que têm valor objetivo z. A Figura 29.2(b) mostra as retas $x_1 + x_2 = 0$, $x_1 + x_2 = 4$ e $x_1 + x_2 = 8$. Como a região viável na Figura 29.2 é limitada, deve existir algum valor máximo z para o qual a interseção da reta $x_1 + x_2 = z$ com a região viável é não vazia. Qualquer ponto em que isso ocorra é uma solução ótima para o programa linear que, nesse caso, é o ponto $x_1 = 2$ e $x_2 = 6$ com valor objetivo 8.

Não é por acidente que uma solução ótima para o programa linear ocorre em um vértice da região viável. O valor máximo de z para o qual a reta $x_1 + x_2 = z$ intercepta a região viável deve estar no contorno da região viável e, assim, a interseção dessa reta com o contorno da região viável é um vértice único ou um segmento de reta. Se a interseção é um vértice único, existe apenas uma solução ótima, e ela é esse vértice. Se a interseção é um segmento de reta, todo ponto nesse segmento de reta deve ter o mesmo valor objetivo; em particular, ambas as extremidades do segmento de reta são soluções ótimas. Visto que cada extremidade de um segmento de reta é um vértice, também nesse caso existe uma solução ótima em um vértice.

Embora não seja fácil representar em gráficos programas lineares com mais de duas variáveis, a mesma intuição é válida. Se temos três variáveis, então cada restrição corresponde a um semiespaço no espaço tridimensional. A interseção desses semiespaços forma a região viável. O conjunto de pontos para os quais a função objetivo obtém um valor z é agora um plano. Se todos os coeficientes da função objetivo são não negativos e se a origem é uma solução viável para o programa linear, então, à medida que afastamos esse plano da origem em uma direção normal à função objetivo, encontramos pontos de valor objetivo crescente. (Se a origem não é viável ou se alguns coeficientes na função objetivo são negativos, o quadro intuitivo se torna um pouco mais complicado.) Como ocorre em duas dimensões, se a região viável é convexa, o conjunto de pontos que alcançam o valor objetivo ótimo deve incluir um vértice da região viável. De modo semelhante, se temos n variáveis, cada restrição define um semiespaço no espaço n dimensional. A região viável formada pela interseção desses semiespaços é denominada ***simplex****. A função objetivo é agora um hiperplano e, devido à convexidade, uma solução ótima ainda ocorre em um vértice da simplex.

O ***algoritmo simplex*** toma como entrada um programa linear e retorna uma solução ótima. Começa em algum vértice do simplex e executa uma sequência de iterações. Em cada iteração, o algoritmo percorre uma aresta do simplex de um vértice atual até um vértice vizinho cujo valor objetivo não é menor que o do vértice atual (e normalmente é maior). O algoritmo simplex termina quando atinge um máximo local, que é um vértice em relação ao qual todos os vértices vizinhos têm um valor objetivo menor. Como a região viável é convexa e a função objetivo é linear, esse local ótimo é, na verdade, um ótimo global. Na Seção 29.4, usaremos um conceito denominado “dualidade” para mostrar que a solução devolvida pelo algoritmo simplex é de fato ótima.

Embora a visão geométrica dê uma boa perspectiva intuitiva para as operações do algoritmo simplex, não faremos referência explícita a ela quando desenvolvermos os detalhes do algoritmo simplex na Seção 29.3. Em vez disso, adotaremos uma visão algébrica. Primeiro, escrevemos o programa linear dado em forma de folgas, que é um conjunto de igualdades lineares. Essas igualdades lineares expressam algumas das variáveis, denominadas “variáveis básicas”, em termos de outras variáveis, denominadas “variáveis não básicas”. Passamos de um vértice para outro fazendo uma variável básica se tornar não básica e fazendo uma variável não básica se tornar básica. Essa operação é denominada “pivô” e, de um ponto de vista algébrico, nada mais é do que reescrever o programa linear em uma forma de folgas equivalente.

O exemplo de duas variáveis que já citamos foi particularmente simples. Precisaremos abordar muitos outros detalhes neste capítulo. Essas questões incluem identificar programas lineares que não têm nenhuma solução, programas lineares que não têm nenhuma solução ótima finita e programas lineares para os quais a origem não é uma solução viável.

Aplicações de programação linear

A programação linear tem grande número de aplicações. Qualquer livro didático sobre pesquisa operacional está repleto de exemplos de programação linear, e a programação linear tornou-se uma ferramenta-padrão ensinada a alunos

na maioria dos cursos de administração de empresas. O cenário eleitoral é um exemplo típico. Dois outros exemplos de programação linear são os seguintes:

- Uma empresa aérea deseja programar as tripulações de seus voos. A Federal Aviation Administration impõe muitas restrições, como limitar o número de horas consecutivas que cada membro da tripulação pode trabalhar e insistir que uma determinada tripulação trabalhe somente em um modelo de aeronave por mês. A empresa aérea quer programar tripulações em todos os seus voos usando o menor número possível de tripulantes.
- Uma empresa petrolífera quer decidir onde perfurar em busca de petróleo. A instalação de uma plataforma de perfuração em determinado local tem um custo associado e, com base em pesquisas geológicas, um retorno esperado de algum número de barris de petróleo. A empresa tem um orçamento limitado para localizar novas áreas de perfuração e quer maximizar a quantidade de petróleo que espera encontrar, dado esse orçamento.

Usamos programas lineares também para modelar e resolver problemas e combinatórios, como os que aparecem neste livro. Já vimos um caso especial de programação linear usado para resolver sistemas de restrições de diferenças sobre grafos na Seção 24.4. Na Seção 29.2 estudaremos como formular vários problemas de grafos e redes de fluxo como programas lineares. Na Seção 35.4, utilizaremos programação linear como ferramenta para determinar uma solução aproximada para um outro problema de grafos.

Algoritmos para programação linear

Este capítulo estuda o algoritmo simplex. Esse algoritmo, quando implementado cuidadosamente, muitas vezes, resolve programas lineares gerais rapidamente na prática. Porém, com algumas entradas criadas cuidadosamente, o algoritmo simplex pode exigir tempo exponencial. O primeiro algoritmo de tempo polinomial para programação linear foi o ***algoritmo dos elipsóides***, cuja execução é lenta na prática. Uma segunda classe de algoritmos de tempo polinomial é conhecida como ***métodos de pontos interiores***. Ao contrário do algoritmo simplex, que percorre o exterior da região viável e mantém uma solução viável que é um vértice do simplex em cada iteração, esses algoritmos percorrem o interior da região viável. As soluções intermediárias, embora viáveis, não são necessariamente vértices do simplex, mas a solução final é um vértice. Para entradas grandes, a execução de algoritmos de pontos interiores pode ser tão rápida quanto a do algoritmo simplex e, às vezes, mais rápida.

Se acrescentarmos a um programa linear o requisito adicional de que todas as variáveis devem ter valores inteiros, temos um ***programa linear inteiro***. O Exercício 34.5-3 pede que você mostre que só determinar uma solução viável para esse problema já é NP-difícil; visto que não há nenhum algoritmo de tempo polinomial conhecido para quaisquer problemas NP-difíceis, não existe nenhum algoritmo de tempo polinomial conhecido para programação linear inteira. Ao contrário, podemos resolver um problema geral de programação linear em tempo polinomial.

Neste capítulo, se tivermos um programa linear com variáveis $x = (x_1, x_2, ..., x_n)$ e quisermos nos referir a um valor particular das variáveis, usaremos a notação $x = (x_1, x_2, \ldots, x_n)$.

29.1 FORMA-PADRÃO E FORMA DE FOLGAS

Esta seção descreve dois formatos, a forma-padrão e a forma de folgas, que são úteis quando especificamos e trabalhamos com programas lineares. Na forma-padrão todas as restrições são desigualdades, enquanto na forma de folgas todas as restrições são igualdades (exceto as que exigem que as variáveis sejam não negativas).

Forma-padrão

Na ***forma-padrão***, temos n números *reais* $c_1, c_2, ..., c_n$, m números reais $b_1, b_2, ..., b_m$ e mn números reais a_{ij} para $i = 1, 2, ..., m$ e $j = 1, 2, ..., n$. Desejamos determinar n números reais $x_1, x_2, ..., x_n$ que

$$\text{maximizem} \quad \sum_{j=1}^{n} c_j x_j \tag{29.16}$$

sujeito a

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad \text{para } i = 1, 2, \ldots, m \tag{29.17}$$

$$x_j \geq 0 \qquad \text{para } j = 1, 2, \ldots, n\,. \tag{29.18}$$

Generalizando a terminologia que apresentamos para o programa linear de duas variáveis, denominamos a expressão (29.16) ***função objetivo*** e as $n + m$ desigualdades nas linhas (29.17) e (29.18) ***restrições***. As n restrições na linha (29.18) são as ***restrições de não negatividade***. Um programa linear arbitrário não precisa ter restrições de não negatividade, mas a forma-padrão as exige. Às vezes, achamos que é conveniente expressar um programa linear de uma forma mais compacta. Se criarmos uma matriz $m \times n$ $A = (a_{ij})$, um m-vetor $b = (b_i)$, um n-vetor $c = (c_j)$ e um n-vetor $x = (x_j)$, então podemos reescrever o programa linear definido em (29.16)-(29.18) como

$$\text{maximizar} \quad c^T x \tag{29.19}$$

sujeito a

$$Ax \leq b \tag{29.20}$$

$$x \geq 0. \tag{29.21}$$

Na reta (29.19), $c_T x$ é o produto interno de dois vetores. Na desigualdade (29.20), Ax é um produto vetor-matriz e, na desigualdade (29.21), $x \geq 0$ significa que cada entrada do vetor x deve ser não negativa. Vemos que podemos especificar um programa linear na forma-padrão por uma tupla (A, b, c) e adotaremos a convenção que A, b e c sempre têm as dimensões dadas acima.

Agora, apresentamos a terminologia para descrever soluções para programas lineares. Usamos parte dessa terminologia no exemplo anterior de um programa linear com duas variáveis. Denominamos ***solução viável*** uma configuração das variáveis x que satisfazem todas as restrições, enquanto uma configuração das variáveis x que deixa de satisfazer no mínimo uma restrição é uma ***solução inviável***. Dizemos que uma solução x tem ***valor objetivo*** c_T x**.** Uma ***solução viável*** x cujo valor objetivo é máximo para todas as soluções viáveis é uma ***solução ótima***, e denominamos seu valor objetivo c_T x ***valor objetivo ótimo***. Se um programa linear não tem nenhuma solução viável, dizemos que o programa linear é ***inviável***; caso contrário, é ***viável***. Se um programa linear tem algumas soluções viáveis mas não tem um valor objetivo ótimo finito, dizemos que o programa linear é ***ilimitado***. O Exercício 29.1-9 pede que você mostre que um programa linear pode ter um valor objetivo ótimo finito mesmo que a região viável não seja limitada.

Converter programas lineares para a forma-padrão

Sempre é possível converter um programa linear, dado como minimizador ou maximizador de uma função linear sujeita a restrições lineares, para a forma-padrão. Um programa linear pode não estar na forma-padrão por uma das quatro razões possíveis:

1. A função objetivo pode ser uma minimização em vez de uma maximização.
2. Pode haver variáveis sem restrições de não negatividade.
3. Pode haver ***restrições de igualdade*** que têm um sinal de igualdade em vez de um sinal de menor que ou igual a.
4. Pode haver ***restrições de desigualdade*** mas, em vez de terem um sinal de menor que ou igual a, elas têm um sinal de maior que ou igual a.

Ao converter um programa linear L em outro programa linear L', gostaríamos que a propriedade de uma solução ótima para L' produzir uma solução ótima para L. Para captar essa ideia, dizemos que dois programas lineares de maximização L e L' são ***equivalentes*** se, para cada solução viável para L com valor objetivo z, existe uma solução

viável correspondente x' para L' com valor objetivo z e, para cada solução viável x' para L' com valor objetivo z, existe uma solução viável x correspondente para L com valor objetivo z. (Essa definição não implica uma correspondência unívoca entre soluções viáveis.) Um programa linear de minimização L e um programa linear de maximização L' são equivalentes se, para cada solução viável x para L com valor objetivo z, existe uma solução viável correspondente x' para L' com valor objetivo $-z$ e, para cada solução x' viável para L' com valor objetivo z, existe uma solução viável x correspondente para L com valor objetivo $-z$.

Mostramos agora como eliminar, um a um, cada um dos problemas possíveis da lista apresentada no início desta subseção. Depois de eliminar cada um deles, demonstraremos que o novo programa linear é equivalente ao antigo.

Para converter um programa linear de minimização L em um programa linear de maximização L' equivalente, simplesmente negamos os coeficientes na função objetivo. Visto que L e L' têm conjuntos idênticos de soluções viáveis e que, para qualquer solução viável, o valor objetivo em L é o negativo do valor objetivo em L', esses dois programas lineares são equivalentes. Por exemplo, se temos o programa linear

$$\begin{array}{lrcrcl}
\text{minimizar} & -2x_1 & + & 3x_2 & & \\
\text{sujeito a} & & & & & \\
 & x_1 & + & x_2 & = & 7 \\
 & x_1 & - & 2x_2 & \leq & 4 \\
 & x_1 & & & \geq & 0.
\end{array}$$

e negamos os coeficientes da função objetivo, obtemos

$$\begin{array}{lrcrcl}
\text{maximizar} & 2x_1 & - & 3x_2 & & \\
\text{sujeito a} & & & & & \\
 & x_1 & + & x_2 & = & 7 \\
 & x_1 & - & 2x_2 & \leq & 4 \\
 & x_1 & & & \geq & 0.
\end{array}$$

Em seguida, mostramos como converter um programa linear no qual algumas das variáveis não têm restrições de não negatividade em um programa no qual cada variável tem uma restrição de não negatividade. Suponha que alguma variável x_j não tenha uma restrição de não negatividade. Então, substituímos cada ocorrência de x_j por $x'_j - x''_j$, e adicionamos as restrições de não negatividade $x'_j \geq 0$ e $x''_j \geq 0$. Assim, se a função objetivo tem um termo $c_j x_j$, nós o substituímos por $c_j x'_j - c_j x''_j$ e, se a restrição i tem um termo $a_{ij} x_j$, nós o substituímos por $a_{ij} x'_j - a_{ij} x''_j$. Qualquer solução viável $\hat{x}$ para o novo programa linear corresponde a uma solução viável para o programa linear original x com $x = \hat{x}' - \hat{x}''$ e com o mesmo valor objetivo. Além disso, qualquer solução viável x para o programa linear original corresponde a uma solução viável $\hat{x}$ para o novo programa linear com $\hat{x}'_j = x_j$ e $\hat{x}''_j = 0$ se $x \geq 0$ ou com $\hat{x}''_j = x_J$ e $\hat{x}'_j = 0$ se $x < 0$. Os dois programas lineares têm o mesmo valor objetivo independentemente do sinal de x_j. Assim, os dois programas lineares são equivalentes. Aplicamos esse esquema de conversão a cada variável que não tem uma restrição de não negatividade para produzir um programa linear equivalente no qual todas as variáveis têm restrições de não negatividade.

Continuando o exemplo, queremos garantir que cada variável tenha uma restrição de não negatividade correspondente. A variável x_1 tem tal restrição, mas a variável x_2 não tem. Portanto, substituímos x_2 por duas variáveis x'_2 e x''_2, e modificamos o programa linear para obter

$$
\begin{array}{llllllll}
\text{maximizar} & 2x_1 & - & 3x'_2 & + & 3x''_2 & & \\
\text{sujeito a} & & & & & & & \\
& x_1 & + & x'_2 & - & x''_2 & = & 7 \\
& x_1 & - & 2x'_2 & + & 2x''_2 & \leq & 4 \\
& x_1, x'_2, x''_2 & & & & & \geq & 0.
\end{array}
\qquad (29.22)
$$

Em seguida, convertemos as restrições de igualdade em restrições de desigualdade. Suponha que um programa linear tenha uma restrição de igualdade $f(x_1, x_2, ..., x_n) = b$. Visto que $x = y$ se e somente se $x \geq y$ e $x \leq y$, podemos substituir essa restrição de igualdade pelo par de restrições de desigualdade $f(x_1, x_2, ..., x_n) \leq b$ e $f(x_1, x_2, ..., x_n) \geq b$. Repetindo essa conversão para cada restrição de igualdade, temos um programa linear no qual todas as restrições são desigualdades.

Finalmente, podemos converter as restrições maior que ou igual a em restrições menor que ou igual a, multiplicando todas essas restrições por -1. Isto é, qualquer desigualdade da forma

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$$

é equivalente a

$$\sum_{j=1}^{n} -a_{ij} x_j \leq -b_i \,.$$

Assim, substituindo cada coeficiente a_{ij} por $-a_{ij}$ e cada valor b_i por $-b_i$, obtemos uma restrição menor que ou igual a equivalente.

Concluindo nosso exemplo, substituímos a igualdade na restrição (29.22) por duas desigualdades, obtendo

$$
\begin{array}{llllllll}
\text{maximizar} & 2x_1 & - & 3x'_2 & + & 3x''_2 & & \\
\text{sujeito a} & & & & & & & \\
& x_1 & + & x'_2 & - & x''_2 & \leq & 7 \\
& x_1 & + & x'_2 & - & x''_2 & \geq & 7 \\
& x_1 & - & 2x'_2 & + & 2x''_2 & \leq & 4 \\
& x_1, x'_2, x''_2 & & & & & \geq & 0.
\end{array}
\qquad (29.23)
$$

Finalmente, negamos a restrição (29.23). Para manter a consistência para os nomes de variáveis, renomeamos x'_2 como x_2 e x''_2 como x_3, e obtemos a forma-padrão

$$
\begin{array}{llllllll}
\text{maximizar} & 2x_1 & - & 3x_2 & + & 3x_3 & & \qquad (29.24) \\
\text{sujeito a} & & & & & & & \\
& x_1 & + & x_2 & - & x_3 & \leq & 7 \qquad (29.25) \\
& -x_1 & - & x_2 & + & x_3 & \leq & -7 \qquad (29.26) \\
& x_1 & - & 2x_2 & + & 2x_3 & \leq & 4 \qquad (29.27) \\
& x_1, x_2, x_3 & & & & & \geq & 0. \qquad (29.28)
\end{array}
$$

Converter programas lineares para a forma de folgas

Para resolver eficientemente um programa linear com o algoritmo simplex, preferimos expressá-lo em uma forma na qual algumas das restrições são restrições de igualdade. Mais exatamente, nós o converteremos para uma forma na qual as restrições de não negatividade são as únicas restrições de desigualdade, e as restrições restantes são igualdades. Seja

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad (29.29)$$

uma restrição de desigualdade. Introduzimos uma nova variável s e reescrevemos a desigualdade (29.29) como as duas restrições

$$s = b_i - \sum_{j=1}^{n} a_{ij} x_j \qquad (29.30)$$

$$s \geq 0. \qquad (29.31)$$

Denominamos s ***variável de folga*** porque ela mede a ***folga***, ou diferença, entre o lado esquerdo e o lado direito da equação (29.29). (Veremos em breve por que achamos que é conveniente escrever a restrição com apenas a variável de folga no lado esquerdo.) Como a desigualdade (29.29) é verdadeira se e somente se a equação (29.30) e a desigualdade (29.31) são verdadeiras, podemos converter cada restrição de desigualdade de um programa linear desse modo para obter um programa linear equivalente no qual as únicas restrições de desigualdade são as restrições de não negatividade. Quando convertermos da forma-padrão para a forma de folgas, usaremos x_{n+i} (em vez de s) para denotar a variável de folga associada à i-ésima desigualdade. Portanto, a i-ésima restrição é

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j , \qquad (29.32)$$

juntamente com a restrição de não negatividade $x_{n+i} \geq 0$.

Convertendo cada restrição de um programa linear na forma-padrão, obtemos um programa linear em uma forma diferente. Por exemplo, para o programa linear descrito em (29.24)- (29.28), introduzimos variáveis de folga x_4, x_5 e x_6, obtendo

$$\begin{array}{llr} \text{maximizar} & 2x_1 - 3x_2 + 3x_3 & (29.33) \\ \text{sujeito a} & & \\ & x_4 = 7 - x_1 - x_2 + x_3 & (29.34) \\ & x_5 = -7 + x_1 + x_2 - x_3 & (29.35) \\ & x_6 = 4 - x_1 + 2x_2 - 2x_3 & (29.36) \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0. & (29.37) \end{array}$$

Nesse programa linear, todas as restrições exceto as restrições de não negatividade são igualdades e cada variável está sujeita a uma restrição de não negatividade. Escrevemos cada restrição de igualdade com uma das variáveis no lado esquerdo da igualdade e todas as outras no lado direito. Além disso, cada equação tem o mesmo conjunto de variáveis no lado direito, e essas variáveis são também as únicas que aparecem na função objetivo. As variáveis no lado esquerdo das igualdades são denominadas ***variáveis básicas***, e as do lado direito são denominadas ***variáveis não básicas***.

Para programas lineares que satisfazem essas condições, às vezes omitimos as palavras "maximizar" e "sujeito a", bem como as restrições explícitas de não negatividade. Utilizaremos também a variável z para denotar o valor da função objetivo. Denominamos o formato resultante ***forma de folgas***. Se escrevermos o programa linear dado em (29.33)-(29.37) em forma de folgas, obteremos

$$\begin{array}{lr} z = 2x_1 - 3x_2 + 3x_3 & (29.38) \\ x_4 = 7 - x_1 - x_2 + x_3 & (29.39) \\ x_5 = -7 + x_1 + x_2 - x_3 & (29.40) \\ x_6 = 4 - x_1 + 2x_2 - 2x_3 & (29.41) \end{array}$$

Como ocorre com a forma-padrão, achamos que seja conveniente adotar uma notação mais concisa para descrever uma forma de folgas. Como veremos na Seção 29.3, os conjuntos de variáveis básicas e não básicas mudarão à medida que o algoritmo simplex for executado. Usamos N para denotar o conjunto de índices das variáveis não básicas e B para denotar o conjunto de índices das variáveis básicas. Sempre temos que $|N| = n$, $|B| = m$ e $N \cup B = \{1, 2, ..., n + m\}$. As equações são indexadas pelas entradas de B e as variáveis no lado direito são indexadas pelas

entradas de N. Como na forma-padrão, usamos b_i, c_j e a_{ij} para denotar termos constantes e coeficientes. Também usamos v para denotar um termo constante opcional na função objetivo. (Um pouco mais adiante veremos que incluir o termo constante na função objetivo facilita a determinação do valor da função objetivo.) Assim, podemos definir concisamente uma forma de folgas por uma tupla (N, B, A, b, c, v) denotando a forma de folgas

$$z = v + \sum_{j \in N} c_j x_j \tag{29.42}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{para } i \in B, \tag{29.43}$$

na qual todas as variáveis x são restritas a ser não negativas. Como subtraímos a soma $\sum_{j \in N} a_{ij} x_j$ em (29.43), os valores a_{ij} são, na verdade, os negativos dos coeficientes que "aparecem" na forma de folgas.

Por exemplo, na forma de folgas

$$\begin{aligned}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2},
\end{aligned}$$

temos $B = \{1, 2, 4\}$, $N = \{3, 5, 6\}$,

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix}.$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix},$$

$c = (c_3\ c_5\ c_6)^T = (-1/6\ -1/6\ -2/3)^T$ e $v = 28$. Observe que os índices em A, b e c não são necessariamente conjuntos de inteiros contíguos; eles dependem dos conjuntos de índices B e N. Como exemplo de que as entradas de A são os negativos dos coeficientes que aparecem na forma de folgas, observe que a equação para x_1 inclui o termo $x_3/6$, ainda que o coeficiente a_{13} seja realmente -1/6 em vez de +1/6.

Exercícios

29.1-1 Se expressarmos o programa linear em (29.24)-(29.28) na notação compacta de (29.19)-(29.21), o que são n, m, A, b e c?

29.1-2 Dê três soluções viáveis para o programa linear em (29.24)-(29.28). Qual é o valor objetivo de cada uma?

29.1-3 Para a forma de folgas em (29.38)-(29.41), o que são N, B, A, b, c e v?

29.1-4 Converta o seguinte programa linear para a forma-padrão:

$$\begin{array}{lrcrcrcr}
\text{minimizar} & 2x_1 & + & 7x_2 & + & x_3 & & \\
\text{sujeito a} & & & & & & & \\
& x_1 & & & - & x_3 & = & 7 \\
& 3x_1 & + & x_2 & & & \geq & 24 \\
& & & x_2 & & & \geq & 0 \\
& & & & & x_3 & \leq & 0.
\end{array}$$

29.1-5 Converta o seguinte programa linear para a forma de folgas:

$$\begin{array}{lrcrcrcr}
\text{maximizar} & 2x_1 & & & - & 6x_3 & & \\
\text{sujeito a} & & & & & & & \\
& x_1 & + & x_2 & - & x_3 & \leq & 7 \\
& 3x_1 & - & x_2 & & & \geq & 8 \\
& -x_1 & + & 2x_2 & + & 2x_3 & \geq & 0 \\
& & & x_1, x_2, x_3 & & & \geq & 0.
\end{array}$$

Quais são as variáveis básicas e não básicas?

29.1-6 Mostre que o seguinte programa linear é inviável:

$$\begin{array}{lrcrcr}
\text{maximizar} & 3x_1 & - & 2x_2 & & \\
\text{sujeito a} & & & & & \\
& x_1 & + & x_2 & \leq & 2 \\
& -2x_1 & - & 2x_2 & \leq & -10 \\
& & & x_1, x_2 & \geq & 0.
\end{array}$$

29.1-7 Mostre que o seguinte programa linear é ilimitado:

$$\begin{array}{lrcrcr} \text{maximizar} & x_1 & - & x_2 & & \\ \text{sujeito a} & & & & & \\ & -2x_1 & + & x_2 & \leq & -1 \\ & -x_1 & - & 2x_2 & \leq & -2 \\ & & x_1, x_2 & & \geq & 0. \end{array}$$

29.1-8 Suponha que tenhamos um programa linear geral com n variáveis e m restrições, e suponha que o convertemos para a forma-padrão. Dê um limite superior para o número de variáveis e restrições no programa linear resultante.

29.1-9 Dê um exemplo de programa linear para o qual a região viável não é limitada, mas o valor objetivo ótimo é finito.

29.2 FORMULAR PROBLEMAS COMO PROGRAMAS LINEARES

Embora o foco deste capítulo seja o algoritmo simplex, também é importante que saibamos reconhecer quando podemos formular um problema como um programa linear. Uma vez expresso o problema como um programa linear de tamanho polinomial, podemos resolvê-lo em tempo polinomial pelo algoritmo dos elipsoides ou dos pontos interiores. Vários pacotes de software de programação linear podem resolver problemas eficientemente; portanto, uma vez expresso como um programa linear, tal pacote pode resolvê-lo.

Examinaremos vários exemplos concretos de problemas de programação linear. Começamos com dois problemas que já estudamos: o problema de caminhos mínimos de fonte única (veja o Capítulo 24) e o problema de fluxo máximo (veja o Capítulo 26). Então, descrevemos o problema de fluxo de custo mínimo. Embora o problema de fluxo de custo mínimo tenha um algoritmo de tempo polinomial que não se baseia em programação linear, não o descreveremos. Finalmente, descrevemos o problema do fluxo de várias mercadorias, para o qual o único algoritmo de tempo polinomial conhecido se baseia em programação linear.

Quando resolvemos problemas de grafos na Parte VI, usamos notação de atributo, tal como $v.d$ e $(u, v).f$. Todavia, problemas lineares normalmente usam variáveis com índices em vez de objetos com atributos anexos. Portanto, quando expressarmos variáveis em programas lineares, indicaremos vértices e arestas por índices. Por exemplo, não denotamos o peso do caminho mínimo para o vértice por v por $v.d$, mas por d_v. De modo semelhante, não denotamos o fluxo do vértice u ao vértice v por $(u, v).f$, mas por f_{uv}. Quando se trata de quantidades dadas como entradas para problemas, por exemplo, capacidades ou pesos de arestas, continuaremos a usar notações como $w(u, v)$ e $c(u,v)$.

Caminhos mínimos

Podemos formular o problema de caminhos mínimos de fonte única como um programa linear. Nesta seção, focalizaremos a formulação do problema de caminhos mínimos para um par, deixando a extensão para o problema mais geral de caminhos mínimos de fonte única para o Exercício 29.2-3.

No problema de caminhos mínimos para um par, temos um grafo ponderado dirigido $G = (V, E)$, com função peso $w : E \rightarrow$ que mapeia arestas para pesos de valor real, um vértice de fonte s e um vértice de destino t. Desejamos calcular o valor d_t, que é o peso de um caminho mínimo de s a t. Para expressar esse problema como um programa linear, precisamos determinar um conjunto de variáveis e restrições que definem quando temos um caminho mínimo desde s a t. Felizmente, o algoritmo de Bellman-Ford faz exatamente isso. Quando termina, o algoritmo de Bellman-

Ford calculou, para cada vértice v, um valor d_v (usando agora a notação por índices em vez da notação por atributo) tal que para cada aresta $(u, v) \in E$, temos $d_v + d_u + w(u, v)$. Inicialmente, o vértice de fonte recebe um valor $d_s = 0$, que nunca muda. Assim, obtemos o seguinte programa linear para calcular o peso do caminho mínimo de s a t:

$$
\begin{array}{llllr}
\text{maximizar} & d_t & & & (29.44) \\
\text{sujeito a} & & & & \\
& d_v \leq & d_u + w(u,v) & \text{para cada aresta } (u,v) \in E\,, & (29.45) \\
& d_s = & 0. & & (29.46)
\end{array}
$$

Você talvez se surpreenda porque esse programa linear maximiza uma função objetivo quando supostamente deveria calcular caminhos mínimos. Não queremos minimizar a função objetivo, visto que definir $d_v = 0$ para todo $v \in V$ produziria uma solução ótima para o programa linear sem resolver o problema dos caminhos mínimos. Maximizamos porque uma solução ótima para o problema dos caminhos mínimos define cada d_v como $\min_{u:(u,v)\in E}\{d_u + w(u, v)\}$, de modo que d_v é o maior valor que é menor ou igual a todos os valores no conjunto $\{d_u + w(u, v)\}$. Queremos maximizar d_v para todos os vértices v em um caminho mínimo de s a t sujeito a essas restrições em todos os vértices v, e maximizar d_t cumpre essa meta.

Esse programa linear tem $|V|$ variáveis d_v, uma para cada vértice $v \in V$. Há também $|E| + 1$ restrições, uma para cada aresta, mais a restrição adicional de que o peso do caminho mínimo do vértice fonte tem o valor 0.

Fluxo máximo

Em seguida, expressamos o problema do fluxo máximo como um programa linear. Lembre-se de que temos um grafo dirigido $G = (V, E)$ no qual cada aresta $(u, v) \in E$ tem uma capacidade não negativa $c(u, v) \geq 0$, e dois vértices preeminentes, uma fonte s e um sorvedouro t. Como definimos na Seção 26.1, um fluxo é uma função não negativa de valor real $f : V \times V \rightarrow$ que satisfaz a restrição de capacidade e conservação de fluxo. Um fluxo máximo é um fluxo que satisfaz essas restrições e maximiza o valor de fluxo, que é o fluxo total que sai da fonte menos o fluxo total que entra na fonte. Portanto, um fluxo satisfaz restrições lineares, e o valor de um fluxo é uma função linear. Lembrando também que convencionamos que $c(u, v) = 0$ se $(u, v) \notin E$ e que não há arestas antiparalelas, podemos expressar o problema de fluxo máximo como um programa linear:

$$
\begin{array}{llr}
\text{maximizar} \quad \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} & & (29.47) \\
\text{sujeito a} & & \\
\quad f_{uv} \leq c(u,v) & \text{para cada } u, v \in V\,, & (29.48) \\
\quad \sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} & \text{para cada } u \in V - \{s, t\}, & (29.49) \\
\quad f_{uv} \geq 0 & \text{para cada } u, v \in \text{V}. & (29.50)
\end{array}
$$

Esse programa linear tem $|V|^2$ variáveis, correspondendo ao fluxo entre cada par de vértices, e $2|V|^2 + |V| - 2$ restrições.

Normalmente, é mais eficiente resolver um programa linear de tamanho menor. O programa linear em (29.47)-(29.50) tem, por facilidade de notação, fluxo e capacidade 0 para cada par de vértices u, v com $(u, v) \notin E$. Seria mais eficiente reescrever o programa linear de modo que ele tenha $O(V + E)$ restrições. O Exercício (29.2-5) pede que você faça isso.

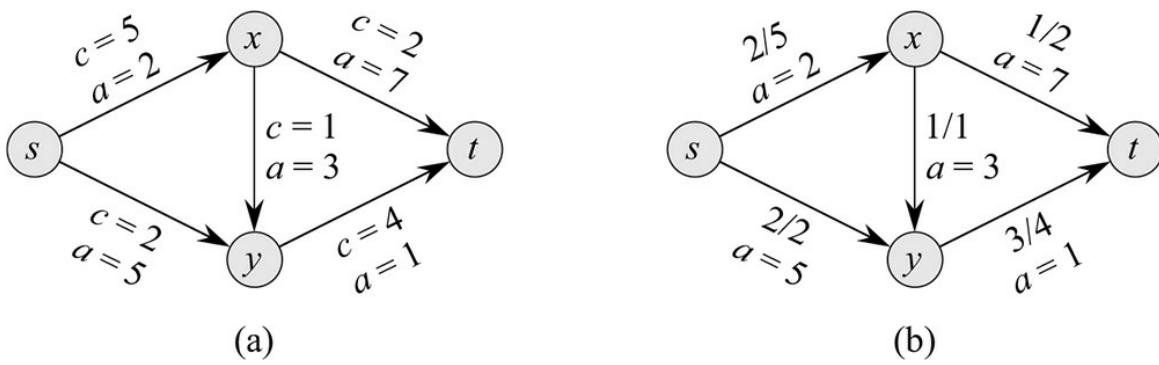
Fluxo de custo mínimo

Nesta seção, usamos programação linear para resolver problemas para os quais já conhecíamos algoritmos eficientes. Na verdade, muitas vezes, um algoritmo eficiente projetado especificamente para um problema, como o

algoritmo de Dijkstra para o problema de caminhos mínimos de fonte única ou o método push-relabel para fluxo máximo, será mais eficiente que a programação linear, tanto na teoria quanto na prática.

O real poder da programação linear vem da capacidade de resolver novos problemas. Lembre-se do problema enfrentado pelo político no início deste capítulo. O problema de obter um número suficiente de votos e ao mesmo tempo não gastar muito dinheiro não é resolvido por qualquer dos algoritmos que estudamos neste livro; no entanto, podemos resolvê-lo por programação linear. Há muitos livros que descrevem tais problemas reais que a programação linear pode resolver. A programação linear também é particularmente útil para resolver variantes de problemas para os quais talvez ainda não conheçamos um algoritmo eficiente.

Considere, por exemplo, a seguinte generalização do problema de fluxo máximo. Suponha que, além de uma capacidade $c(u, v)$ para cada aresta (u, v) tenhamos um custo de valor real $a(u, v)$. Como no problema do fluxo máximo, convencionamos que $c(u, v) = 0$ se $(u, v) \notin E$ e que não existe nenhuma aresta antiparalela. Se enviarmos f_{uv} unidades de fluxo pela aresta (u, v), incorreremos em um custo $a(u, v)f_{uv}$. Temos também uma demanda de fluxo d. Desejamos enviar d unidades de fluxo de s a t e ao mesmo tempo minimizar o custo total $\sum_{(u,v)\in E} a(u, v) f_{uv}$, incorrido pelo fluxo. Esse problema é conhecido como ***problema de fluxo de custo mínimo***.

A Figura 29.3(a) mostra um exemplo do problema de fluxo de custo mínimo. Desejamos enviar quatro unidades de fluxo de s para t, incorrendo no custo total mínimo. Qualquer fluxo válido específico, isto é, uma função f que satisfaça as restrições (29.48)-(29.50), incorre em um custo total

$\sum_{(u,v)\in E} a(u, v) f_{uv}$. Desejamos determinar o fluxo específico de quatro unidades que minimiza esse custo. A Figura 29.3(b) mostra uma solução ótima, com custo total $\sum_{(u,v)\in E} a(u, v) f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$.

Existem algoritmos de tempo polinomial projetados especificamente para o problema de fluxo de custo mínimo, mas eles não estão no escopo deste livro. Entretanto, podemos expressar o problema de fluxo de custo mínimo como um programa linear. O programa linear é semelhante ao do problema de fluxo máximo, com a restrição adicional que o valor do fluxo deve ser exatamente d unidades e com a nova função objetivo de minimizar o custo:

$$\text{minimizar} \quad \sum_{(u,v)\in E} a(u,v) f_{vu} \tag{29.51}$$

sujeito a

$$\begin{aligned} f_{uv} &\leq c(u,v) && \text{para cada } u, v \in V, \\ \sum_{v\in E} f_{vu} - \sum_{v\in V} f_{uv} &= 0 && \text{para cada } u \in V - \{s, t\}, \\ \sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} &= d, \\ f_{uv} &\geq 0 && \text{para cada } u, v \in V. \end{aligned} \tag{29.52}$$

Figura 29.3 (**a**) Exemplo de um problema de fluxo de custo mínimo. Denotamos as capacidades por c e os custos por a. O vértice s é a fonte e o vértice t é o sorvedouro, e desejamos enviar quatro unidades de fluxo de s a t. (**b**) Solução para o problema de fluxo de custo

mínimo no qual quatro unidades de fluxo são enviadas de s a t. Para cada aresta, o fluxo e a capacidade estão escritos como fluxo/capacidade.

Fluxo multimercadorias

Como exemplo final, consideramos outro problema de fluxo. Suponha que a empresa Lucky Puck da Seção 26.1 decida diversificar sua linha de produtos e comercializar bastões e capacetes de hóquei, além de discos de hóquei. Cada item de equipamento é produzido em sua própria fábrica, tem seu próprio armazém e deve ser transportado todo dia da fábrica ao armazém. Os bastões são fabricados em Vancouver e devem ser transportados para Saskatoon, e os capacetes são fabricados em Edmonton e devem ser transportados para Regina. No entanto, a capacidade da rede de transporte não muda, e os diferentes itens, ou ***mercadorias***, devem compartilhar a mesma rede.

Esse exemplo é uma instância de um ***problema de fluxo multimercadorias***. Nesse problema, temos novamente um grafo dirigido $G = (V, E)$ no qual cada aresta $(u, v) \in E$ tem uma capacidade não negativa $c(u, v) \geq 0$. Como no problema de fluxo máximo, supomos implicitamente que $c(u, v) = 0$ para $(u, v) \notin E$ e que o grafo não tem nenhuma aresta antiparalela. Além disso, temos k mercadorias diferentes, $K_1, K_2, ..., K_k$, onde especificamos a mercadoria i pela tripla $K_i = (s_i, t_i, d_i)$. Aqui, o vértice s_i é a fonte da mercadoria i, o vértice t_i é o sorvedouro da mercadoria i e d_i é a demanda para a mercadoria i, que é o valor de fluxo desejado para a mercadoria de s_i a t_i. Definimos um fluxo para a mercadoria i, denotado por f_i (de modo que f_{iuv} é o fluxo da mercadoria i do vértice u ao vértice v) como uma função de valor real que satisfaz as restrições de conservação de fluxo e capacidade. Agora definimos f_{uv}, o ***fluxo agregado***, como a soma dos vários fluxos de mercadorias, de modo que $f_{uv}\ \sum_{i=1}^{k} f_{iuv}$. O fluxo agregado na aresta (u, v) não deve ser maior que a capacidade da aresta (u, v). Não estamos tentando minimizar qualquer função objetivo nesse problema; precisamos somente determinar se tal fluxo existe. Assim, escrevemos um programa linear com uma função objetivo "nula":

$$\begin{array}{llll}
\text{minimizar} & & 0 & \\
\text{sujeito a} & & & \\
& \displaystyle\sum_{i=1}^{k} f_{iuv} \leq & c(u,v) & \text{para cada } u, v \in V, \\
& \displaystyle\sum_{v \in E} f_{iuv} - \sum_{v \in V} f_{ivu} = & 0 & \text{para cada } i = 1, 2, \ldots, k \text{ e} \\
& & & \text{para cada } u \in V - \{s_i, t_i\}, \\
& \displaystyle\sum_{v \in V} f_{i,s_i,v} - \sum_{v \in V} f_{i,v,s_i} = & d_i & \text{para cada } i = 1, 2, \ldots, k, \\
& f_{iuv} \geq & 0 & \text{para cada } u, v \in V \text{ e} \\
& & & \text{para cada } i = 1, 2, \ldots, k.
\end{array}$$

O único algoritmo de tempo polinomial conhecido para esse problema o expressa como um programa linear e depois o resolve com um algoritmo de programação linear de tempo polinomial.

Exercícios

29.2-1 Expresse o programa linear de caminhos mínimos para um par de (29.44)-(29.46) na forma-padrão.

29.2-2 Escreva explicitamente o programa linear correspondente a determinar o caminho mínimo do nó s ao nó y na Figura 24.2(a).

29.2-3 No problema de caminhos mínimos de fonte única, queremos determinar os pesos de caminhos mínimos de um vértice de fonte s a todos os vértices $v \in V$. Dado um grafo G, escreva um programa linear para o qual a solução tenha a seguinte propriedade: d_v é o peso do caminho mínimo de s a v para cada vértice $v \in V$.

29.2-4 Escreva explicitamente o programa linear correspondente a determinar o fluxo máximo na Figura 26.1(a).

29.2-5 Reescreva o programa linear para fluxo máximo (29.47)-(29.50) de modo que ele use apenas $O(V + E)$ restrições.

29.2-6 Escreva um programa linear que, dado um grafo bipartido $G = (V, E)$, resolva o problema do emparelhamento máximo em grafo bipartido.

29.2-7 No ***problema do fluxo de custo mínimo para várias mercadorias***, temos um grafo dirigido $G = (V, E)$ no qual cada aresta $(u, v) \in E$ tem uma capacidade não negativa $c(u, v) \geq 0$ e um custo $a(u, v)$. Como no problema de fluxo de várias mercadorias, temos k mercadorias diferentes, $K_1, K_2, ..., K_k$, onde especificamos a mercadoria i pela tripla $K_i = (s_i, t_i, d_i)$. Definimos o fluxo f_i para a mercadoria i e o fluxo agregado $f_{u,v}$ na aresta (u, v) como no problema de fluxo de várias mercadorias. Um fluxo viável é aquele no qual o fluxo agregado em cada aresta (u, v) não é maior que a capacidade da aresta (u, v). O custo de um fluxo é $\sum_{u,v \in V} a(u, v) f_{uv}$, e a meta é determinar o fluxo viável de custo mínimo. Expresse esse problema como um programa linear.

29.3 O ALGORITMO SIMPLEX

O algoritmo simplex é o método clássico para resolver programas lineares. Ao contrário da maioria dos outros algoritmos neste livro, seu tempo de execução não é polinomial no pior caso, porém sugere ideias para programas lineares e, muitas vezes, é extraordinariamente rápido na prática.

Além de ter uma interpretação geométrica, já descrita neste capítulo, o algoritmo simplex guarda alguma semelhança com o método de eliminação de Gauss, discutido na Seção 28.1. O método de eliminação de Gauss começa com um sistema de igualdades lineares cuja solução é desconhecida. Em cada iteração, reescrevemos esse sistema em uma forma equivalente que tem alguma estrutura adicional. Após um certo número de iterações, reescrevemos o sistema de modo que a solução seja simples de obter. O algoritmo simplex age de maneira semelhante, e podemos considerá-lo como o método de eliminação de Gauss para desigualdades.

Agora, descrevemos a ideia principal por trás de uma iteração do algoritmo simplex. Associada a cada iteração haverá "uma solução básica" fácil de obter pela forma de folgas do programa linear: definir cada variável não básica como 0 e calcular os valores das variáveis básicas pelas restrições de igualdade. Uma iteração converte uma forma de folgas em uma forma de folgas equivalente. O valor objetivo da solução básica viável associada não será menor que o da iteração anterior e, em geral, será maior. Para conseguir esse aumento no valor objetivo, escolhemos uma variável não básica tal que, se fôssemos aumentar o valor dessa variável começando em 0, o valor objetivo também aumentaria. O valor do aumento que podemos conseguir para a variável está limitado pelas outras restrições. Em particular, só o aumentamos até que algumas variáveis básicas se tornem 0. Então reescrevemos a forma de folgas, trocando os papéis daquela variável básica e da variável não básica escolhida. Embora tenhamos usado uma configuração específica das variáveis para guiar o algoritmo, a qual usaremos em nossas provas, o algoritmo não mantém explicitamente essa solução. Ele simplesmente reescreve o programa linear até que uma solução ótima se torne "óbvia".

Um exemplo do algoritmo simplex

Começamos com um exemplo estendido. Considere o seguinte programa linear na forma-padrão:

$$\begin{array}{llr} \text{maximizar} & 3x_1 + x_2 + 2x_3 & (29.53) \\ \text{sujeito a} & & \\ & x_1 + x_2 + 3x_3 \leq 30 & (29.54) \\ & 2x_1 + 2x_2 + 5x_3 \leq 24 & (29.55) \\ & 4x_1 + x_2 + 2x_3 \leq 36 & (29.56) \\ & x_1, x_2, x_3 \geq 0. & (29.57) \end{array}$$

Para usar o algoritmo simplex, devemos converter o programa linear para a forma de folgas; vimos como fazer isso na Seção 29.1. Além de ser uma manipulação algébrica, a folga é um conceito algorítmico útil. Lembre-se de que, na Seção 29.1, observamos que cada variável tem uma restrição de não negatividade correspondente; dizemos que uma restrição de igualdade é ***exata*** para uma configuração particular de suas variáveis não básicas se ela faz com que a variável básica da restrição se torne 0. De modo semelhante, uma configuração das variáveis não básicas que tornasse uma variável básica negativa ***viola*** aquela restrição. Assim, as variáveis de folga mantêm explicitamente o quanto cada restrição está longe de ser exata e, portanto, ajudam a determinar o quanto podemos aumentar valores de variáveis não básicas sem violar nenhuma restrição.

Associando as variáveis de folga x_4, x_5 e x_6 respectivamente às desigualdades (29.54)-(29.56) e, expressando o programa linear em forma de folgas, obtemos

$$\begin{array}{lr} z = 3x_1 + x_2 + 2x_3 & (29.58) \\ x_4 = 30 - x_1 - x_2 - 3x_3 & (29.59) \\ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 & (29.60) \\ x_6 = 36 - 4x_1 - x_2 - 2x_3. & (29.61) \end{array}$$

O sistema de restrições (29.59)-(29.61) tem três equações e seis variáveis. Qualquer configuração das variáveis x_1, x_2 e x_3 define valores para x_4, x_5 e x_6; portanto, temos um número infinito de soluções para esse sistema de equações. Uma solução é viável se todos os valores $x_1, x_2, ..., x_6$ são não negativos, e pode haver também um número infinito de soluções viáveis. O número infinito de soluções possíveis para um sistema como esse será útil em provas posteriores. Focalizamos a ***solução básica***: faça todas as variáveis (não básicas) no lado direito iguais a 0 e depois calcule os valores das variáveis (básicas) no lado esquerdo. Nesse exemplo, a solução básica é $(x_1, x_2, ..., x_6) = (0, 0, 0, 30, 24, 36)$, e ela tem valor objetivo $z = (3 \cdot 0) + (1 \cdot 0) + (2 \cdot 0) = 0$. Observe que essa solução básica define $x_i = b_i$ para cada $i \in B$. Uma iteração do algoritmo simplex reescreve o conjunto de equações e a função objetivo de modo a colocar um conjunto diferente de variáveis no lado direito. Assim, uma solução básica diferente é associada ao problema reescrito. Enfatizamos que reescrever não altera de modo algum o problema de programação linear subjacente; o problema em uma iteração tem o conjunto de soluções viáveis idêntico ao do problema na iteração anterior. Entretanto, o problema realmente tem uma solução básica diferente da solução na iteração anterior.

Se uma solução básica também é viável, nós a denominamos ***solução básica viável***. Durante a execução do algoritmo simplex, a solução básica quase sempre é uma solução básica viável. Porém, veremos na Seção 29.5 que, para as primeiras iterações do algoritmo simplex, a solução básica pode não ser viável.

Nossa meta, em cada iteração, é reformular o programa linear de modo que a solução básica tenha um valor objetivo maior. Selecionamos uma variável não básica x_e cujo coeficiente na função objetivo é positivo e aumentamos o valor de x_e o quanto possível, sem violar quaisquer das restrições. A variável x_e se torna básica, e alguma outra variável x_l se torna não básica. Os valores de outras variáveis básicas e da função objetivo também podem mudar.

Para continuar o exemplo, vamos pensar em aumentar o valor de x_1. À medida que aumentamos x_1, os valores de x_4, x_5 e x_6 diminuem. Como temos uma restrição de não negatividade para cada variável, não podemos permitir que nenhuma delas se torne negativa. Se x_1 aumenta acima de 30, então x_4 se torna negativa, e x_5 e x_6 se tornam negativas quando x_1 aumenta acima de 12 e 9, respectivamente. A terceira restrição (29.61) é a mais rígida e limita o quanto podemos aumentar x_1. Portanto, trocamos os papéis de x_1 e x_6. Resolvemos a equação (29.61) para x_1 e obtemos

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}. \qquad (29.62)$$

Para reescrever as outras equações com x_6 no lado direito, substituímos x_1 usando a equação (29.62). Fazendo isso para a equação (29.59), obtemos

$$\begin{aligned} x_4 &= 30 - x_1 - x_2 - 3x_3 \\ &= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{4} - \frac{x_6}{4}\right) - x_2 - 3x_3 \\ &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} - \frac{x_6}{4}. \end{aligned} \tag{29.63}$$

De modo semelhante,combinamos a equação (29.62) com a restrição (29.60) e com a função objetivo (29.58) para reescrever nosso programa linear na seguinte forma:

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{16} \tag{29.64}$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \tag{29.65}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \tag{29.66}$$

$$x_4 = 6 - \frac{3x_2}{2} - 4x_5 + \frac{x_6}{2}. \tag{29.67}$$

Denominamos essa operação ***pivô***. Como acabamos de demonstrar, um pivô escolhe uma variável não básica x_e, denominada ***variável de entrada***, e uma variável básica x_l, denominada ***variável de saída***, e troca suas funções.

O programa linear descrito nas equações (29.64)-(29.77) é equivalente ao programa linear descrito nas equações (29.58)-(29.61). Executamos duas operações no algoritmo simplex: reescrever equações de modo que as variáveis passem do lado esquerdo para o lado direito e substituir uma equação em outra. A primeira operação cria trivialmente um problema equivalente, e a segunda, por álgebra linear elementar, também cria um problema equivalente (veja o Exercício 29.3-3).

Para demonstrar essa equivalência, observe que nossa solução básica original (0, 0, 0, 30, 24, 36) satisfaz as novas equações (29.65)-(29.67) e tem valor objetivo 27 + (1/4) · 0 + (1/2) · 0 - (3/4) · 36 = 0. A solução básica associada ao novo programa linear define os valores não básicos como 0, e é (9, 0, 0, 21, 6, 0), com valor objetivo z = 27. Simples aritmética verifica que essa solução também satisfaz as equações (29.59)-(29.61) e, quando substituída na função objetivo (29. 58), tem valor objetivo (3 · 9) + (1 · 0) + (2 · 0) = 27.

Continuando o exemplo, desejamos encontrar uma nova variável cujo valor queremos aumentar. Não queremos aumentar x_6 já que, à medida que seu valor aumenta, o valor objetivo diminui. Podemos tentar aumentar ou x_2 ou x_3; escolhemos x_3. Ate quanto podemos aumentar x_3 sem violar nenhuma das restrições? A restrição (29.65) a limita a 18, a restrição (29.66) a limita a 42/5 e a restrição (29.67) a limita a 3/2. A terceira restrição é novamente a mais rígida e, portanto, reescrevemos a terceira restrição de modo que x_3 fique no lado esquerdo e x_5 no lado direito. Então, substituímos essa nova equação, $x_3 = 3/2 - 3x_2/8 - x_5/4 + x_6/8$, nas equações (29.64)- (29.66) e obtemos o sistema novo, mas equivalente

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \tag{29.68}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \tag{29.69}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_3}{4} + \frac{x_6}{8} \quad (29.70)$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}. \quad (29.71)$$

Esse sistema tem a solução básica associada (33/4, 0, 3/2, 69/4, 0, 0), com valor objetivo 111/4. Agora o único modo de aumentar o valor objetivo é aumentar x_2. As três restrições dão limites superiores de 132, 4 e ∞, respectivamente. (Obtemos um limite superior ∞ pela restrição (29.71) porque, à medida que aumentamos x_2, o valor da variável básica x_4 também aumenta. Portanto, essa restrição não limita o quanto podemos aumentar x_2.) Aumentamos x_2 para 4, e ela se torna não básica. Então resolvemos a equação (29.70) para x_2 e a substituímos nas outras equações para obter

$$z = 28 - \frac{x_3}{6} - \frac{x_5}{3} - \frac{2x_6}{3} \quad (29.72)$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \quad (29.73)$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \quad (29.74)$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2}. \quad (29.75)$$

Nesse ponto, todos os coeficientes na função objetivo são negativos. Como veremos mais adiante neste capítulo, essa situação ocorre somente quando reescrevemos o programa linear de modo que a solução básica seja uma solução ótima. Assim, para esse problema, a solução (8, 4, 0, 18, 0, 0), com valor objetivo 28, é ótima. Agora podemos retornar ao nosso programa linear original dado em (29.53)-(29.57). As únicas variáveis no programa linear original são x_1, x_2 e x_3, e assim nossa solução é $x_1 = 8$, $x_2 = 4$ e $x_3 = 0$, com valor objetivo $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Observe que os valores das variáveis de folga na solução final medem quanta folga sobra em cada desigualdade. A variável de folga x_4 é 18 e, na desigualdade (29.57), o lado esquerdo, com valor $8 + 4 + 0 = 12$, é 18 unidades menor que o lado direito 30. As variáveis de folga x_5 e x_6 são 0 e, de fato, nas desigualdades (29.55) e (29.56), o lado esquerdo e o lado direito são iguais. Observe também que, embora os coeficientes na forma original de folgas sejam inteiros, os coeficientes nos outros programas lineares não são necessariamente inteiros, e as soluções intermediárias não são necessariamente inteiras. Além disso, a solução final para um programa linear não precisa ser inteira; é simples coincidência que esse exemplo tenha uma solução inteira.

Pivotar

Agora, formalizamos o procedimento para pivotar. O procedimento PIVOT toma como entrada uma forma de folgas, dada pela tupla (N, B, A, b, c, v), o índice l da variável de saída x_l, e o índice e da variável de entrada x_e. Ele retorna a tupla $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$ que descreve a nova forma de folgas. (Lembre-se mais uma vez de que as entradas das matrizes $m \times n$ A e $\hat{A}$ são na realidade as negativas dos coeficientes que aparecem na forma de folgas.)

```
PIVOT(N, B, A, b, c, v, l, e)
1    // Calcule os coeficientes para nova variável básica x_e.
2    seja Â uma nova matriz m × n
3    b̂_e = b_l / a_le
4    for cada j ∈ N − {e}
5         â_ej / a_lj/a_le
6    â_el = 1/a_le
7    // Calcular os coeficientes das restrições restantes.
8    for cada i ∈ B − {l}
9         b̂_i = b_i − a_ie b̂_e
10        for cada j ∈ N − {e}
11             â_ij = a_ij − a_ie â_ej
12        â_il = −a_ie â_el
13   // Calcular a função objetivo.
14   v̂ = v + c_e b̂_e
15   for cada j ∈ N − {e}
16        ĉ_j = c_j − c_e â_ej
17   ĉ_l = −c_e â_el
18   // Calcular novos conjuntos de variáveis básicas e não básicas.
19   N̂ = N − {e} ∪ {l}
20   B̂ = B − {l} ∪ {e}
21   return (N̂, B̂, Â, b̂, ĉ, v̂)
```

PIVOT funciona da maneira descrita a seguir. As linhas 3-6 calculam os coeficientes na nova equação para x_e reescrevendo a equação que tem x_1 no lado esquerdo de modo que agora tenha x_e no lado esquerdo. As linhas 8-12 atualizam as equações restantes substituindo o lado direito dessa nova equação em cada ocorrência de x_e. As linhas 14-17 fazem a mesma substituição para a função objetivo, e as linhas 19 e 20 atualizam os conjuntos de variáveis não básicas e básicas. A linha 21 retorna a nova forma de folgas. Como dado, se $a_{le} = 0$, PIVOT causaria um erro de divisão por zero, mas, como veremos nas provas dos Lemas 29.2 e 29.12, chamamos PIVOT somente quando $a_{le} \neq 0$.

Agora, resumimos o efeito de PIVOT sobre os valores das variáveis na solução básica.

Lema 29.1

Considere uma chamada PIVOT(N, B, A, b, c, v, l, e) na qual $a_{le} \neq 0$. Sejam ($\hat{N}$, $\hat{B}$, $\hat{A}$, $\hat{b}$, $\hat{c}$, $\hat{v}$) os valores retornados pela chamada e seja x a solução básica depois da chamada. Então

1. $\bar{x}_j = 0$ para cada $j \in \hat{N}$.
2. $\bar{x}_e = b_l / a_{le}$.
3. $\bar{x}_i = b_i - a_{ie}\hat{b}_e$ para cada $i \in \hat{B} - \{e\}$.

Prova A primeira afirmação é verdadeira porque a solução básica sempre define todas as variáveis não básicas como 0. Quando definimos cada variável não básica como 0 em uma restrição

$$x_i = \hat{b}_i - \sum_{j \in \hat{N}} \hat{a}_{ij} x_j$$

temos que $x_i = \hat{b}_i$ para cada $i \in \hat{B}$. Visto que $e \in \hat{B}$, a linha 3 de PIVOT dá

$$\bar{x}_e = \hat{b}_e = b_l / a_{le}$$

o que prova a segunda afirmação. De modo semelhante, usando a linha 9 para cada $i \in \hat{B} - \{e\}$, temos

$$\bar{x}_i = \hat{b}_i = b_i - a_{ie}\hat{b}_e ,$$

o que prova a terceira afirmação.

O algoritmo simplex formal

Agora estamos prontos para formalizar o algoritmo simplex, que já demonstramos como exemplo. Tal exemplo foi particularmente interessante, e poderíamos ter várias outras questões a abordar:

- Como determinamos se um programa linear é viável?

- O que fazemos se o programa linear é viável, mas a solução básica inicial não é viável?
- Como determinamos se um programa linear é ilimitado?
- Como escolhemos as variáveis que entram e saem?

Na Seção 29.5, mostraremos como determinar se um problema é viável e, se for, como determinar uma forma de folgas na qual a solução básica inicial seja viável. Portanto, vamos supor que temos um procedimento INITIALIZE-SIMPLEX(A, b, c) que toma como entrada um programa linear na forma-padrão, isto é, uma matriz $m \times n$ $A = (a_{ij})$, um m– vetor $b = (b_i)$ e um n– vetor $c = (c_j)$. Se o problema for inviável, o procedimento retorna uma mensagem indicando que o programa é inviável e em seguida termina. Caso contrário, retorna uma forma de folgas para a qual a solução básica inicial é viável.

O procedimento SIMPLEX toma como entrada um programa linear na forma-padrão, exatamente como foi descrito e retorna um n– vetor $x = (x_j)$ que é uma solução ótima para o programa linear descrito em (29.19)-(29.21).

```
SIMPLEX(A, b, c)
1    (N,B,A, b, c, v) = INITIALIZE-SIMPLEX(A, b, c)
2    seja Δ um novo vetor de comprimento m
3    while algum índice j ∈ N tem c_j > 0
4        escolher um índice e ∈ N para o qual c_e > 0
5        for cada índice i ∈ B
6            if a_ie > 0
7                Δ_i = b_i/a_ie
8            else Δ_i = ∞,
9        escolher um índice l ∈ B que minimiza Δ_l
10       if Δ_l == ∞
11           return "ilimitado"
12       else (N,B,A, b, c, v) = PIVOT(N,B,A, b, c, v, l, e)
13   for i = 1 to n
14       if i ∈ B
15           x̄_i = b_i
16       else x̄_i = 0
17   return (x̄_1, x̄_2, … , x̄_n)
```

O procedimento SIMPLEX funciona da seguinte maneira: na linha 1, chama o procedimento INITIALIZE-SIMPLEX(A, b, c) já descrito, que determina que o programa linear é inviável ou retorna uma forma de folgas, para a qual a solução básica é viável. O laço **while** das linhas 3-12 forma a parte principal do algoritmo. Se todos os coeficientes da função objetivo

são negativos, o laço **while** termina. Caso contrário, a linha 4 seleciona como variável de entrada uma variável x_e cujo coeficiente na função objetivo é positivo. Embora possamos escolher qualquer variável como variável de entrada, supomos que usamos alguma regra determinística previamente especificada. Em seguida, as linhas 5-9 verificam cada restrição e escolhem a que impõe limites mais estritos ao aumento possível de x_e sem violar nenhuma das restrições de não negatividade; a variável básica associada a essa restrição é x_l. Novamente, estamos livres para escolher uma de diversas variáveis para sair da base, mas supomos que usamos alguma regra determinística previamente especificada. Se nenhuma das restrições impuser um limite ao aumento possível da variável de entrada, o algoritmo retornará "ilimitado" na linha 11. Caso contrário, a linha 12 trocará os papéis da variável que entra com a que sai, chamando PIVOT(N, B, A, b, c, v, l, e), como já descrevemos. As linhas 13-16 calculam uma solução x_1, x_2, … , x_n para as variáveis de programação linear originais definindo todas as variáveis não básicas como 0 e cada variável básica x_i como b_i, e a linha 17 devolve esses valores.

Para mostrar que SIMPLEX é correto, primeiro mostramos que, se SIMPLEX tem uma solução inicial viável e eventualmente termina, então ele retorna uma solução viável ou determina que o programa linear é ilimitado. Então, mostramos que SIMPLEX termina. Finalmente, na Seção 29.4 (Teorema 29.10), mostramos que a solução retornada é ótima.

Lema 29.2

Dado um programa linear (A, b, c), suponha que a chamada INITIALIZE-SIMPLEX na linha 1 de SIMPLEX retorne uma forma de folgas para a qual a solução básica é viável. Então, se SIMPLEX retorna uma solução na linha 17, essa solução é uma solução viável para o programa linear. Se SIMPLEX retorna "ilimitado" na linha 11, o programa linear é ilimitado.

Prova Usamos o seguinte invariante de laço de três partes:

No início de cada iteração do laço **while** das linhas 3-12 ,

1. A forma de folgas é equivalente à forma de folgas retornada pela chamada de INITIALIZE-SIMPLEX.
2. Para cada $i \in B$, temos $b_i \geq 0$.
3. A solução básica associada à forma de folgas é viável.

Inicialização: A equivalência das formas de folgas é trivial para a primeira iteração. Admitimos, no enunciado do lema, que a chamada a INITIALIZE-SIMPLEX na linha 1 de SIMPLEX retorna uma forma de folgas para a qual a solução básica é viável. Assim, a terceira parte do invariante é verdadeira. Além disso, visto que a solução básica define cada variável básica x_i como b_i , temos que $b_i \geq 0$ para todo $i \in B$. Assim, a segunda parte do invariante é válida.

Manutenção: Mostraremos que cada iteração do laço **while** mantém o invariante de laço, supondo que a declaração **return** na linha 11 não é executada. Trataremos o caso no qual a linha 11 é executada quando discutirmos término. Uma iteração do laço **while** troca o papel de uma variável básica e uma variável não básica chamando o procedimento PIVOT. Pelo Exercício 29.3-3, a forma de folgas é equivalente à da iteração anterior que, pelo invariante de laço, é equivalente à forma de folgas inicial. Agora demonstramos a segunda parte do invariante de laço. Supomos que, no início de cada iteração do laço **while**, $b_i \geq 0$ para cada $i \in B$, e mostraremos que essas desigualdades permanecem verdadeiras após a chamada a PIVOT na linha 12. Visto que as únicas mudanças para as variáveis b_i e o conjunto B de variáveis básicas ocorrem nessa atribuição, basta mostrar que a linha 12 mantém essa parte do invariante. Fazemos b_i, a_{ij} e B se referirem a valores antes da chamada de PIVOT, e se referir a valores retornados de PIVOT.

Primeiro, observamos que $\hat{b}_e \geq 0$ porque $b_l \geq 0$ pelo invariante de laço, $a_{le} > 0$ pelas linhas 6 e 9 de SIMPLEX e $\hat{b}_e = b_l / a_{le}$ pela linha 3 de PIVOT.

Para os índices restantes $i \in B - \{l\}$, temos que

$$\begin{aligned}\hat{b}_i &= b_i - a_{ie}\hat{b}_e && \text{(pela linha 9 de Pivot)}\\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(pela linha 3 de Pivot).} \qquad (29.76)\end{aligned}$$

Temos dois casos a considerar, dependendo de $a_{ie} > 0$ ou $a_{ie} \leq 0$. Se $a_{ie} > 0$, então, desde que escolhamos l tal que

$$b_l/a_{le} \leq b_i/a_{ie} \text{ para todo } i \in B, \qquad (29.77)$$

temos

$$\begin{aligned}\hat{b}_i &= b_i - a_{ie}(b_l/a_{le}) && \text{(pela equação (29.76))}\\ &\geq b_i - a_{ie}(b_i/a_{ie}) && \text{(pela desigualdade (29.77))}\\ &= b_i - b_i\\ &= 0,\end{aligned}$$

e, assim, $\hat{b}_i \geq 0$. Se $a_{ie} \leq 0$, então como a_{le}, b_i e b_l são não negativos, a equação (29.76) implica que $\hat{b}_i$ também deve ser não negativo.

Agora demonstramos que a solução básica é viável, isto é, que todas as variáveis têm valores não negativos. As variáveis não básicas são atribuídas 0 e, portanto, são não negativas. Cada variável básica x_i é definida pela equação

$$x_i = b_i - \sum_{j \in N}^{n} a_{ij}x_j \cdot$$

A solução básica define $x_i = b_i$. Usando a segunda parte do invariante de laço, concluímos que cada variável básica é não negativa.

Término: O laço **while** pode terminar de dois modos. Se terminar por causa da condição na linha 3, a solução básica atual é viável e a linha 17 retorna essa solução. O outro modo de terminar é retornar "ilimitado" na linha 11. Nesse caso, para cada iteração do laço **for** nas linhas 5-8, quando a linha 6 é executada verificamos que $a_{ie} \leq 0$. Considere a solução x definida como

$$\bar{x}_i = \begin{cases}\infty & \text{se } i = e,\\ 0 & \text{se } i \in N - \{e\},\\ b_i - \sum_{j \in N} a_{ij}\bar{x}_j & \text{se } i \in B.\end{cases}$$

Agora mostramos que essa solução é viável, isto é, que todas as variáveis são não negativas. As variáveis não básicas exceto x_e são 0, e $x_e = \infty > 0$; assim, todas as variáveis não básicas são não negativas. Para cada variável básica x_i, temos

$$\begin{aligned}\hat{x}_i &= b_i - \sum_{j \in N} a_{ij}\bar{x}_j\\ &= b_i - a_{ie}\bar{x}_e \cdot\end{aligned}$$

O invariante de laço implica que $b_i \geq 0$, e temos $a_{ie} \leq 0$ e $x_e = \infty > 0$. Portanto, $x_i \geq 0$. Mostramos agora que o valor objetivo para a solução é ilimitado. O valor objetivo é

$$
\begin{aligned}
z &= v + \sum_{j \in N} c_j \overline{x}_j \\
&= v + c_e \overline{x}_e \cdot
\end{aligned}
$$

Visto que $c_e > 0$ (pela linha 4de SIMPLEX) e $x_e = \infty$, o valor objetivo é ∞, e assim o programa linear é ilimitado.

Resta mostrar que SIMPLEX termina e, quando termina, a solução que ele retorna é ótima. A Seção 29.4 tratará de otimalidade. Agora, discutimos término.

Término

No exemplo dado no início desta seção, cada iteração do algoritmo simplex aumentou o valor objetivo associado à solução básica. Como o Exercício 29.3-2 pede que você mostre, nenhuma iteração de SIMPLEX pode diminuir o valor objetivo associado à solução básica. Infelizmente, é possível que uma iteração deixe o valor objetivo inalterado. Esse fenômeno é denominado ***degenerescência*** e agora o estudaremos em detalhes.

A atribuição na linha 14 de PIVOT, $\hat{v} = v + c_e \hat{b}_e$, muda o valor objetivo. Visto que SIMPLEX chama PIVOT somente quando $c_e > 0$, o único modo de o valor objetivo permanecer inalterado (isto é, $\hat{v} = v$) é $\hat{b}$, ser 0. Esse valor é atribuído como $\hat{b}_e = b_l / a_{le}$ na linha 3 de PIVOT. Como sempre chamamos PIVOT com $a_{le} \neq 0$, vemos que, para $\hat{b}_e$ ser igual a 0 e, consequentemente, o valor objetivo permanecer inalterado, devemos ter $b_l = 0$.

De fato, essa situação pode ocorrer. Considere o programa linear

$$
\begin{aligned}
z &= \phantom{8 - {}} x_1 + x_2 + x_3 \\
x_4 &= 8 - x_1 - x_2 \\
x_5 &= \phantom{8 - x_1 - {}} x_2 - x_3 .
\end{aligned}
$$

Suponha que escolhemos x_1 como a variável de entrada e x_4 como a variável de saída. Após pivotar, obtemos

$$
\begin{aligned}
z &= 8 + x_3 + x_4 \\
x_1 &= 8 - x_2 - x_4 \\
x_5 &= x_2 - x_3 .
\end{aligned}
$$

Nesse ponto, nossa única escolha é pivotar com x_3 entrando e x_5 saindo. Visto que $b_5 = 0$, o valor objetivo de 8 permanece inalterado após o pivotamento:

$$
\begin{aligned}
z &= 8 + x_2 - x_4 - x_5 \\
x_1 &= 8 - x_2 - x_4 \\
x_3 &= x_2 - x_5 .
\end{aligned}
$$

O valor objetivo não mudou, mas nossa forma de folgas mudou. Felizmente, se pivotarmos novamente, com x_2 entrando e x_1 saindo, o valor objetivo aumenta (para 16), e o algoritmo simplex poderá continuar.

Valores degenerados podem impedir que o algoritmo simplex termine porque podem levar a um fenômeno conhecido como ***ciclagem***: as formas de folgas em duas iterações diferentes de SIMPLEX são idênticas. Por causa dos valores degenerados, SIMPLEX poderia escolher uma sequência de operações de pivô que deixam o valor objetivo

inalterado, mas repetem uma forma de folgas dentro da sequência. Visto que SIMPLEX é um algoritmo determinístico, se ele ciclar percorrerá a mesma série de formas de folgas para sempre e nunca terminará.

A ciclagem é a única razão pela qual SIMPLEX poderia não terminar. Para mostrar esse fato, temos de desenvolver algumas ferramentas adicionais.

A cada iteração, SIMPLEX mantém A, b, c e v , além dos conjuntos N e B. Embora seja preciso manter A, b, c e v explicitamente para implementar o algoritmo simplex eficientemente, conseguimos passar sem mantê-los. Em outras palavras, os conjuntos de variáveis básicas e não básicas são suficientes para determinar inequivocamente a forma de folgas. Antes de provar esse fato, provamos uma lema algébrico útil.

Lema 29.3

Seja I um conjunto de índices. Para cada $j \in I$, sejam α_j e β_j números reais e seja x_j uma variável de valor real. Seja γ qualquer número real. Suponha que, para quaisquer configurações de x_j, tenhamos

$$\sum_{j\in I}\alpha_j x_j = \gamma + \sum_{j\in I}\beta_j x_j \cdot \tag{29.78}$$

Então, $\alpha_j = \beta_j$ para cada $j \in I$ e $\gamma = 0$.

Prova Visto que a equação (29.78) é válida para quaisquer valores de x_j, podemos usar valores específicos para tirar conclusões sobre α, β e γ. Se fizermos $x_j = 0$ para cada $j \in I$, concluímos que $\gamma = 0$. Agora, escolha um índice arbitrário $j \in I$, e sejam $x_j = 1$ e $x_k = 0$ para todo $k \neq j$. Então, devemos ter $\alpha_j = \beta_j$. Visto que escolhemos j como qualquer índice em I, concluímos que $\alpha_j = \beta_j$ para cada $j \in I$.

Um programa linear determinado tem muitas formas de folgas diferentes; lembre-se de que cada forma de folgas tem o mesmo conjunto de soluções viáveis e ótimas que o programa linear original. Mostramos agora que a forma de folgas de um programa linear é determinada exclusivamente pelo conjunto de variáveis básicas. Isto é, dado o conjunto de variáveis básicas, uma única forma de forma de folgas (um único conjunto de coeficientes e lados direitos) está associado a essas variáveis básicas.

Lema 29.4

Seja (A, b, c) um programa linear na forma-padrão. Dado um conjunto B de variáveis básicas, a forma de folgas associada é determinada inequivocamente.

Prova Suponha, por contradição, que haja duas formas de folga diferentes com o mesmo conjunto B de variáveis básicas. As formas de folga devem também ter conjuntos idênticos $N = \{1, 2, ..., n + m\} - B$ de variáveis não básicas. Escrevemos a primeira forma de folgas como

$$z = v + \sum_{j\in N} c_j x_j \tag{29.79}$$

$$x_i = b_i - \sum_{j\in N} a_{ij} x_j \qquad \text{para } i \in B\ , \tag{29.80}$$

E a segunda como

$$z = v' + \sum_{j\in N} c'_j x_j \tag{29.81}$$

$$x_i = b'_i - \sum_{j\in N} a'_{ij} x_j \qquad \text{para } i \in B\ . \tag{29.82}$$

Considere o sistema de equações formado subtraindo cada equação na linha (29.82) da equação correspondente na linha (29.80). O sistema resultante é

$$0 = (b_i - b'_i) - \sum_{j \in N} (a_{ij} - a'_{ij}) x_j \qquad \text{para } i \in B$$

ou, de modo equivalente,

$$\sum_{j \in N} (a_{ij} x_j = (b_i - b'_i) + \sum_{j \in N} a'_{ij} x_j \qquad \text{para } i \in B\,.$$

Agora, para cada $i \in B$, aplicamos o Lema 29.3 com $\alpha_j = \alpha_{ij}$, $\beta_j = a'_{ij}$ e $\gamma = b_i - b'_i$ e $I = N$. Visto que $\alpha_j = \beta_j$, temos que $a_{ij} = a'_{ij}$ para cada $j \in N$ e, como $\gamma = 0$, temos que $b_i = b'_i$. Assim, para as duas formas de folga, A e b são idênticos a A' e b'. Usando um argumento semelhante, o Exercício 29.3-1 mostra que também deve ocorrer $c = c'$ e $v = v'$ e, consequentemente, que as formas de folga devem ser idênticas.

Agora, podemos mostrar que ciclagem é a única razão pela qual SIMPLEX poderia não terminar.

Lema 29.5

Se SIMPLEX deixar de terminar em, no máximo, $\binom{n+m}{m}$ iterações, ele ciclará.

Prova Pelo Lema 29.4, o conjunto B de variáveis básicas determina inequivocamente uma forma de folgas. Há $n + m$ variáveis e $|B| = m$; portanto, há no máximo $\binom{n+m}{m}$ maneiras de escolher B. Assim, há apenas $\binom{n+m}{m}$ formas de folga distintas. Então, se SIMPLEX for executado por mais de $\binom{n+m}{m}$ iterações, ele deverá ciclar. ■

Ciclar é teoricamente possível, mas extremamente raro. Podemos evitar que isso ocorra escolhendo as variáveis que entram e saem com um pouco mais de cuidado. Uma opção é perturbar ligeiramente a entrada de modo que seja impossível ter duas soluções com o mesmo valor objetivo. Uma outra opção é desempatar escolhendo sempre a variável com menor índice, uma estratégia conhecida como ***regra de Bland***. Omitimos a prova de que essas estratégias evitam ciclagem.

Lema 29.6

Se as linhas 4 e 9 de SIMPLEX sempre desempatam escolhendo a variável com o menor índice, então SIMPLEX termina.

Concluímos esta seção com o lema a seguir.

Lema 29.7

Desde que INITIALIZE-SIMPLEX retorne uma forma de folgas para a qual a solução básica é viável, SIMPLEX informa que um programa linear é ilimitado ou termina com uma solução viável em, no máximo, $\binom{n+m}{m}$ iterações.

Prova Os Lemas 29.2 e 29.6 mostram que, se INITIALIZE-SIMPLEX retorna uma forma de folgas para a qual a solução básica é viável, SIMPLEX informa que um programa linear é ilimitado ou termina com uma solução viável. Pela contrapositiva do Lema 29.5, se SIMPLEX termina com uma solução viável, então termina em, no máximo, $\binom{n+m}{m}$ iterações. ■

29.3-1 Conclua a prova do Lema 29.4 mostrando que deve ser o caso em que $c = c'$ e $v = v'$. ***29.3-2*** Mostre que a chamada a PIVOT na linha 12 de SIMPLEX nunca diminuirá o valor de v. ***29.3-3*** Prove que a forma de folgas dada ao procedimento PIVOT e a forma de folga que o
procedimento devolve são equivalentes.

29.3-4 Suponha que convertemos um programa linear (A, b, c) em forma-padrão para forma de folgas. Mostre que a solução básica é viável se e somente se $b_i \geq 0$ para $i = 1, 2, ..., m$.

29.3-5 Resolva o seguinte programa linear usando SIMPLEX:

$$\begin{array}{lrcrcr} \text{maximizar} & 18x_1 & + & 12{,}5x_2 & & \\ \text{sujeito a} & & & & & \\ & x_1 & + & x_2 & \leq & 20 \\ & x_1 & & & \leq & 12 \\ & & & x_2 & \leq & 16 \\ & x_1, x_2 & & & \geq & 0. \end{array}$$

29.3-6 Resolva o seguinte programa linear usando SIMPLEX:

$$\begin{array}{lrcrcr} \text{maximizar} & 5x_1 & - & 3x_2 & & \\ \text{sujeito a} & & & & & \\ & x_1 & - & x_2 & \leq & 1 \\ & 2x_1 & + & x_2 & \leq & 2 \\ & x_1, x_2 & & & \geq & 0. \end{array}$$

29.3-7 Resolva o seguinte programa linear usando SIMPLEX:

$$\begin{array}{lrcrcrcr} \text{minimizar} & x_1 & + & x_2 & + & x_3 & & \\ \text{sujeito a} & & & & & & & \\ & 2x_1 & + & 7{,}5x_2 & + & 3x_3 & \geq & 10000 \\ & 20x_1 & + & 5x_2 & + & 10x_3 & \geq & 30000 \\ & x_1, x_2, x_3 & & & & & \geq & 0. \end{array}$$

29.3-8 Na prova do Lema 29.5, argumentamos que há no máximo $\binom{n+m}{m}$ modos de escolher um conjunto B de variáveis básicas. Dê um exemplo de programa linear no qual haja um número estritamente menor que $\binom{n+m}{m}$ modos de escolher o conjunto B.

29.4 DUALIDADE

Provamos que, sob certas circunstâncias, SIMPLEX termina. Contudo, ainda não mostramos que ele realmente encontra uma solução ótima para um programa linear. Para tal, apresentamos um conceito importante, denominado ***dualidade de programação linear***.

A dualidade nos permite provar que uma solução é de fato ótima. Vimos um exemplo de dualidade no Capítulo 26 com o Teorema 26.6, o teorema do fluxo máximo/corte mínimo. Suponha que, dada uma instância de um problema de fluxo máximo, encontramos um fluxo f com valor $|f|$. Como saber se f é um fluxo máximo? Pelo teorema do fluxo máximo/corte mínimo, se pudermos determinar um corte cujo valor também seja $|f|$, teremos verificado que f é de fato um fluxo máximo. Essa relação nos dá um exemplo de dualidade: dado um problema de maximização, definimos um problema de minimização relacionado, tal que os dois problemas têm os mesmos valores objetivos ótimos.

Dado um programa linear no qual o objetivo é maximizar, descreveremos como formular um programa linear ***dual*** no qual o objetivo é minimizar e cujo valor ótimo é idêntico ao do programa linear original. Quando nos referimos a programas lineares duais, denominamos o programa linear original ***primal***.

Dado um programa linear primal na forma-padrão, como em (29.16)-(29.18), definimos o programa linear dual como

$$\text{minimizar} \quad \sum_{i=1}^{m} b_i y_i \tag{29.83}$$

sujeito a

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \quad \text{para } j = 1, 2, \ldots, n, \tag{29.84}$$

$$y_i \geq 0 \quad \text{para } i = 1, 2, \ldots, m. \tag{29.85}$$

Para formar o dual, trocamos a maximização para uma minimização, permutamos os papéis dos coeficientes dos lados direitos e a função objetivo, e substituímos o sinal *menor que ou igual a* por um sinal *maior que ou igual a*. Cada uma das m restrições no primal tem uma variável associada y_i no dual, e cada uma das n restrições no dual tem uma variável x_j associada no primal. Por exemplo, considere o programa linear dado em (29.56)-(29.57). O dual desse programa linear é

$$\text{minimizar} \quad 30y_1 + 24y_2 + 36y_3 \tag{29.86}$$

sujeito a

$$y_1 + 2y_2 + 4y_3 \geq 3 \tag{29.87}$$

$$y_1 + 2y_2 + y_3 \geq 1 \tag{29.88}$$

$$3y_1 + 5y_2 + 2y_3 \geq 2 \tag{29.89}$$

$$y_1, y_2, y_3 \geq 0. \tag{29.90}$$

Mostraremos, no Teorema 29.10, que o valor ótimo do programa linear dual é sempre igual ao valor ótimo do programa linear primal. Além disso, na verdade, o algoritmo simplex resolve implicitamente os programas lineares primal e dual ao mesmo tempo, dando assim uma prova da otimalidade.

Começamos demonstrando a ***dualidade fraca***, que afirma que qualquer solução viável para o programa linear primal tem um valor não maior que o de qualquer solução viável para o programa linear dual.

Lema 29.8 (Dualidade fraca de programação linear)

Seja x uma solução viável para o programa linear primal em (29.16)-(29.18) e seja y solução viável para o programa linear dual em (29.83)-(29.85). Então, temos

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{i=1}^{m} b_i \bar{y}_i \cdot$$

Prova Temos

$$\begin{aligned}\sum_{j=1}^{n} c_j \bar{x}_j &\le \sum_{j=1}^{n}\left(\sum_{i=1}^{m} a_{ij}\bar{y}_i\right)\bar{x}_j && \text{(pelas desigualdades (29.84))}\\ &= \sum_{i=1}^{m}\left(\sum_{j=1}^{n} a_{ij}\bar{x}_j\right)\bar{y}_i\\ &\le \sum_{i=1}^{m} b_i\bar{y}_i && \text{(pelas desigualdades (29.17)).}\end{aligned}$$

Corolário 29.9

Seja x uma solução viável para um programa linear primal (A, b, c) e seja y uma solução viável para o programa linear dual correspondente. Se

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i ,$$

então, x e y são soluções ótimas para os programas lineares primal e dual, respectivamente.

Prova Pelo Lema 29.8, o valor objetivo de uma solução viável para o primal não pode exceder o de uma solução viável para o dual. O programa linear primal é um problema de maximização, e o dual é um problema de minimização. Assim, se as soluções viáveis x e y têm o mesmo valor objetivo, nenhuma delas pode ser melhorada.

Antes de provar que sempre existe uma solução dual cujo valor é igual ao de uma solução primal ótima, descreveremos como encontrar tal solução. Quando executamos o algoritmo simplex no programa linear em (29.53)-(29.57), a iteração final produziu a forma de folgas (29.72)-(29.75) com objetivo $z = 28 - x_3/6 - x_5/6 - 2x_6/3$, $B = \{1, 2, 4\}$ e $N = \{3, 5, 6\}$. Como veremos a seguir, a solução básica associada à forma de folgas final é de fato uma solução ótima para o programa linear; portanto, uma solução ótima para o programa linear (29.53)-(29.57) é = (8, 4, 0), com valor objetivo $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Como também veremos a seguir, podemos ler uma solução dual ótima: os negativos dos coeficientes da função objetivo primal são os valores das variáveis duais. Mais precisamente, suponha que a última forma de folgas da primal seja

$$\begin{aligned} z &= v' + \sum_{j\in N} c'_j x_j \\ x_i &= b'_i - \sum_{j\in N} a'_{ij} x_j \qquad \text{para } i \in B\,. \end{aligned}$$

Então, para produzir uma solução dual ótima, definimos

$$\bar{y}_i = \begin{cases} -c'_{n+i} & \text{se } (n+i) \in N, \\ 0 & \text{otherwise.} \end{cases} \tag{29.91}$$

Assim, uma solução ótima para o programa linear dual definido em (29.86)-(29.90) é $y_1 = 0$ (já que $n + 1 = 4 \in B$), $y_2 = -c'_5 = 1/6$ e $y_3 = -c'_6 = 2/3$. Avaliando a função objetivo dual (29.86), obtemos um valor objetivo de $(30 \cdot 0) + (24 \cdot (1/6)) + (36 \cdot (2/3)) = 28$, o que confirma que o valor objetivo do primal é de fato igual ao valor objetivo do dual. Combinando esses cálculos com o Lema 29.8, obtemos uma prova de que o valor objetivo ótimo do programa linear primal é 28. Agora, mostramos que essa abordagem se aplica em geral: podemos determinar uma solução ótima para o dual e simultaneamente provar que uma solução para o primal é ótima.

Teorema 29.10 (Dualidade de programação linear)

Suponha que SIMPLEX retorne valores $x = (x_1, x_2, \ldots, x_n)$ para o programa linear primal (A, b, c). Sejam N e B as variáveis não básicas e básicas para a forma de folgas final, seja c' a representação dos coeficientes na forma de folgas final e seja $y = (y_1, y_2, \ldots, y_n)$ definido pela equação (29.91). Então, x é uma solução ótima para o programa linear primal, y é uma solução ótima para o programa linear dual e

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i. \tag{29.92}$$

Prova Pelo Corolário 29.9, se podemos determinar soluções viáveis x e N que satisfaçam a equação (29.92), então x e y devem ser soluções primal e dual ótimas. Agora, mostraremos que as soluções x e y descritas no enunciado do teorema satisfazem a equação (29.92).

Suponha que executamos SIMPLEX em um programa linear primal, como dado nas linhas (29.16)-(29.18). O algoritmo prossegue por uma série de formas de folga até terminar com uma forma de folgas final com função objetivo

$$z = v' + \sum_{j \in N} c'_j \bar{x}_j. \tag{29.93}$$

Visto que SIMPLEX terminou com uma solução, pela condição na linha 3 sabemos que

$$c'_j \leq 0 \quad \text{para todo } j \in N. \tag{29.94}$$

Se definirmos

$$c'_j = 0 \quad \text{para todo } j \in B, \tag{29.95}$$

podemos reescrever a equação (29.93) como

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ &= v' + \sum_{j \in N} c'_j x_j + \sum_{j \in B} c'_j x_j && \text{(porque } c'_j = 0 \text{ se } j \in B) \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j. && \text{(porque } N \in B = \{1, 2, \ldots, n+m\}). \end{aligned} \tag{29.96}$$

Para a solução básica x associada a essa forma de folgas final, $x_j = 0$ para todo $j \in N$ e $z = v'$. Visto que todas as formas de folga são equivalentes, se avaliarmos a função objetivo original em x, devemos obter o mesmo valor objetivo, isto é,

$$\begin{aligned} \sum_{j=1}^{n} c_j \bar{x}_j &= v' + \sum_{j=1}^{n+m} c'_j \bar{x}_j && (29.97) \\ &= v' + \sum_{j \in N} c'_j \bar{x}_j + \sum_{j \in B} c'_j \bar{x}_j \\ &= v' + \sum_{j \in N} (c'_j \cdot 0) + \sum_{j \in B} (0 \cdot \bar{x}_j) && (29.98) \\ &= v'. \end{aligned}$$

Agora, mostraremos que $\bar{y}$, definido pela equação (29.91), é viável para o programa linear dual e que seu valor objetivo $\sum_{i=1}^{m} b_i \bar{y}_i$ é igual a $\sum_{j=1}^{n} c_j \bar{x}_j$. A equação (29.97) afirma que a primeira e a última formas de folga, avaliadas em $\bar{x}$, são iguais. De modo mais geral, a equivalência de todas as formas de folga implica que, para *qualquer* conjunto de valores $x = (x_1, x_2, \ldots, x_n)$, temos

$$\sum_{j=1}^{n} c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j \ .$$

Portanto, para qualquer conjunto de valores $x = (x_1, x_2, \ldots, x_n)$, temos

$$\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j & \\
&= v' + \sum_{j=1}^{n+m} c'_j \bar{x}_j \\
&= v' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{j=n+1}^{n+m} c'_j \bar{x}_j \\
&= v' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{i=1}^{m} c'_{n+1} \bar{x}_{n+i} \\
&= v' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{i=1}^{m} (-\bar{y}_i) \bar{x}_{n+i} && \text{(pelas equações (29.91) e (29.95))} \\
&= v' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{i=1}^{m} (-\bar{y}_i) \left(b_i - \sum_{j=1}^{n} a_{ij} \bar{x}_{n+i} \right) && \text{(pela equação (29.32))} \\
&= v' + \sum_{j=1}^{n} c'_j \bar{x}_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{i=1}^{m} \sum_{j=1}^{n} (a_{ij} \bar{x}_j) \bar{y}_i \\
&= v' + \sum_{j=1}^{n} c'_j \bar{x}_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{j=1}^{n} \sum_{i=1}^{m} (a_{ij} \bar{y}_i) \bar{x}_j \\
&= \left(v' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j,
\end{aligned}$$

de modo que

$$\sum_{j=1}^{n} c_j \bar{x}_j = \left(v' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \tag{29.99}$$

Aplicando o Lema 29.3 à equação (29.99), obtemos

$$v' - \sum_{i=1}^{m} b_i \bar{y}_i = 0, \tag{29.100}$$

$$c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i = c_j \text{ para } j = 1, 2, \ldots, n. \tag{29.101}$$

Pela equação (29.100), temos que $\sum_{i=1}^{m} b_i \bar{y}_i = v'$ e, consequentemente, o valor objetivo do dual $\left(\sum_{i=1}^{m} b_i \bar{y}_i \right)$ é igual ao do primal (v'). Resta mostrar que a solução $\bar{y}$ é viável para o problema dual. Pelas desigualdades (29.94) e equação (29.95), temos que $c'_j \leq 0$ para todo $j = 1, 2, \ldots, n + m$. Assim, para qualquer $i = 1, 2, \ldots, m$, as equações (29.101) implicam que

$$\begin{aligned}
c_j &= c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \\
&\leq \sum_{i=1}^{m} a_{ij} \bar{y}_i,
\end{aligned}$$

que satisfaz as restrições (29.84) do dual. Finalmente, visto que $c'_j \leq 0$ para cada $j \in N \cup B$, quando definimos de acordo com a equação (29.91), temos que cada $y_i \geq 0$ e, assim, as restrições de não negatividade também são satisfeitas.

Mostramos que, dado um programa linear viável, se INITIALIZE-SIMPLEX retorna uma solução viável e se SIMPLEX termina sem retornar "ilimitado", a solução retornada é de fato uma solução ótima. Mostramos também como elaborar uma solução ótima para o programa linear dual.

Exercícios

29.4-1 Formule o dual do programa linear dado no Exercício 29.3-4.

29.4-2 Suponha que temos um programa linear que não está na forma-padrão. Poderíamos produzir o dual primeiro convertendo-o para a forma-padrão e depois tomando o dual. Porém, seria mais conveniente poder produzir o dual diretamente. Explique como podemos tomar o dual diretamente de um programa linear arbitrário.

29.4-3 Escreva o dual do programa linear de fluxo máximo, como dado nas linhas (29.47)- (29.50). Explique como interpretar essa formulação como um problema de corte mínimo.

29.4-4 Escreva o dual do programa linear de fluxo de custo mínimo, como dado nas linhas (29.51)-(29.55). Explique como interpretar esse problema em termos de grafos e fluxos.

29.4-5 Mostre que o dual do dual de um programa linear é o programa linear primal.

29.4-6 Qual resultado do Capítulo 26 pode ser interpretado como dualidade fraca para o problema de fluxo máximo?

29.5 A SOLUÇÃO BÁSICA VIÁVEL INICIAL

Nesta seção, primeiro descrevemos como testar se um programa linear é viável e, se for, como produzir uma forma de folgas para a qual a solução básica seja viável. Concluímos provando o teorema fundamental de programação linear, que afirma que o procedimento SIMPLEX sempre produz o resultado correto.

Determinando uma solução inicial

Na Seção 29.3, admitimos que tínhamos um procedimento INITIALIZE-SIMPLEX que determina se um programa linear tem soluções viáveis e, se tiver, dá uma forma de folgas para a qual a solução básica é viável. Descrevemos esse procedimento aqui.

Um programa linear pode ser viável e, apesar disso, a solução básica inicial pode não ser viável. Por exemplo, considere o seguinte programa linear:

$$\text{maximizar} \quad 2x_1 - x_2 \tag{29.102}$$

sujeito a

$$2x_1 - x_2 \leq 2 \tag{29.103}$$

$$x_1 - 5x_2 \leq -4 \tag{29.104}$$

$$x_1, x_2 \geq 0. \tag{29.105}$$

Se tivéssemos de converter esse programa linear para forma de folgas, a solução básica definiria $x_1 = 0$ e $x_2 = 0$. Essa solução infringe a restrição (29.104) e, portanto, não é uma solução viável. Assim, INITIALIZE-SIMPLEX não pode simplesmente retornar a forma de folgas óbvia. Para determinar se um programa linear tem quaisquer soluções viáveis, formularemos um ***programa linear auxiliar***. Por esse programa linear auxiliar, podemos determinar (com um pouco de trabalho) uma forma de folgas para a qual a solução básica seja viável. Além disso, a solução desse programa linear

auxiliar determina se o programa linear inicial é viável e, se for, dará uma solução viável com a qual poderemos inicializar SIMPLEX.

Lema 29.11

Seja L um programa linear na forma-padrão, dado como em (29.16)-(29.18). Seja x_0 uma nova variável e seja L_{aux} o seguinte programa linear com $n + 1$ variáveis:

$$\text{maximizar} \quad -x_0 \tag{29.106}$$

sujeito a

$$\sum_{j=1}^{n} a_{ij}x_j - x_0 \leq b_i \quad \text{para } i = 1, 2, ..., m, \tag{29.107}$$

$$x_j \geq 0 \quad \text{para } j = 0, 1, ..., n. \tag{29.108}$$

Então, L é viável se e somente se o valor objetivo ótimo de L_{aux} é 0.

Prova Suponha que L tenha uma solução viável $x = (x_1, x_2, \ldots, x_n)$. Então, a solução $x_0 = 0$ combinada com x é uma solução viável para L_{aux} com valor objetivo 0. Visto que $x_0 \geq 0$ é uma restrição de L_{aux} e a função objetivo é para maximizar $-x_0$, essa solução deve ser ótima para L_{aux}. Ao contrário, suponha que o valor objetivo ótimo de L_{aux} seja 0. Então, $x_0 = 0$, e os valores das soluções restantes satisfazem as restrições de L.

Agora, descrevemos nossa estratégia para determinar uma solução básica inicial viável para um programa linear L em forma-padrão:

```
INITIALIZE-SIMPLEX(A, b, c)
1     seja k o índice do mínimo b_i
2     if b_k ≥ 0                      // a solução básica inicial é viável?
3          return ({1, 2, ..., n}, {n + 1, n + 2, ..., n + m}, A, b, c, 0)
4     formar L_aux by acrescentando −x_0 ao lado esquerdo de cada restrição e definindo
           a função objetivo como −x_0
5     seja (N,B,A, b, c, v) a forma de folgas resultante para L_aux
6     l = n + k
7     // L_aux tem n + 1 variáveis não básicas e m variáveis básicas
8     (N, B, A, b, c, v) = PIVOT(N, B, A, b, c, v, 1, 0)
9     // A solução básica agora é viável para L_aux.
10    iterar o laço while das linhas 3−12 de SIMPLEX até encontrar uma solução ótima
           para L_aux é encontrado
11    if a solução ótima para L_aux define x̄_0 como 0
12         if x̄_0 é básica
13              execute um pivô (degenerado) para transformá-la em não básica
14         na forma de folgas final de L_aux, eliminar x_0 das restrições e restaurar a função
              objetivo original de L, mas substituir cada variável básica nessa
              função objetivo pelo lado direito da sua restrição associada
15         return a forma de folgas final modificada
16    else return "inviável"
```

INITIALIZE-SIMPLEX funciona da maneira descrita a seguir. Nas linhas 1-3, testamos implicitamente a solução básica para a forma de folgas inicial para L dada por $N = \{1, 2, ..., n\}$, $B = \{n + 1, n + 2, ..., n + m\}$, $x_i = b_i$ para todo $i \in B$ e $x_j = 0$ para todo $j \in N$. (Criar a forma de folgas não requer nenhum esforço explícito, já que os valores de A, b e c são os mesmos na forma de folgas e também na forma-padrão.) Se a linha 2 determinar que essa solução básica é viável — isto é, $x_i \geq 0$ para todo $i \in N \cup B$ —, então a linha 3 retornará a forma de folgas. Caso contrário, na linha 4, formamos o programa linear auxiliar L_{aux} como no Lema 29.11. Visto que a solução básica inicial para L não é viável, a

solução básica inicial para a forma de folgas correspondente a L_{aux} também não é viável. Para encontrar uma solução básica viável, executamos uma única operação de pivô. A linha 6 seleciona $l = n + k$ como o índice da variável básica que será a variável de saída na operação de pivô que será realizada. Visto que as variáveis básicas são $x_{n+1}, x_{n+2}, \ldots,$ x_{n+m}, a variável de saída x_l será aquela que tiver o maior valor negativo. A linha 8 executa aquela chamada de PIVOT, com x_0 entrando e x_l saindo. Em breve veremos que a solução básica resultante dessa chamada de PIVOT será viável. Agora, que temos uma forma de folgas para a qual a solução básica é viável, podemos, na linha 10, chamar repetidamente PIVOT para resolver completamente o programa linear auxiliar. Como o teste da linha 11 demonstra, se encontrarmos uma solução ótima para L_{aux} com valor objetivo 0, então nas linhas 12-14 criaremos uma forma de folgas para L para a qual a solução básica é viável. Para tal, em primeiro lugar, nas linhas 12-13, tratamos o caso degenerado no qual x_0 ainda pode ser básica com valor $x_0 = 0$. Nesse caso, executamos uma etapa de pivô para eliminar x_0 da base usando qualquer $e \in N$ tal que $a_{0e} \neq 0$ como a variável de entrada. A nova solução básica permanece viável; o pivô degenerado não muda o valor de nenhuma das variáveis. Em seguida, eliminamos todos os termos x_0 das restrições e restauramos a função objetivo original para L. A função objetivo original pode conter variáveis básicas e não básicas. Portanto, na função objetivo substituímos cada variável básica pelo lado direito de sua restrição associada. Depois a linha 15 devolve essa forma de folgas modificada. Se, por outro lado, a linha 11 descobre que o programa linear original L é inviável, a linha 16 devolve essa informação.

Agora demonstramos o funcionamento de INITIALIZE-SIMPLEX no programa linear (29.102)- (29.105). Esse programa linear é viável se pudermos determinar valores não negativos para x_1 e x_2 que satisfaçam as desigualdades (29.103) e (29.104). Pelo Lema 29.11, formulamos o programa linear auxiliar

$$\text{maximizar} \quad -x_0 \qquad (29.109)$$

sujeito a

$$\begin{aligned} 2x_1 - x_2 - x_0 &\leq 2 & (29.110)\\ x_1 - 5x_2 - x_0 &\leq -4 & (29.111)\\ x_1, x_2, x_0 &\geq 0. \end{aligned}$$

Pelo Lema 29.11, se o valor objetivo ótimo desse programa linear auxiliar for 0, o programa linear original terá uma solução viável. Se o valor objetivo ótimo desse programa linear auxiliar for negativo, o programa linear original não terá uma solução viável.

Escrevemos esse programa linear na forma de folgas obtendo

$$\begin{aligned} z &= && && && - x_0\\ x_3 &= 2 &&- 2x_1 &&+ x_2 &&+ x_0\\ x_4 &= -4 &&- x_1 &&+ 5x_2 &&+ x_0. \end{aligned}$$

Ainda não estamos fora de perigo porque a solução básica, que definiria $x_4 = -4$, não é viável para esse programa linear auxiliar. Contudo, com uma chamada a PIVOT, podemos converter essa forma de folgas em uma forma na qual a solução básica seja viável. Como a linha 8 indica, escolhemos x_0 como a variável de entrada. Na linha 6, escolhemos como variável de saída x_4, que é a variável básica cujo valor na solução básica é o mais negativo. Depois de pivotar, temos a forma de folgas

$$\begin{aligned} z &= -4 &&- x_1 &&+ 5x_2 &&- x_4\\ x_0 &= 4 &&+ x_1 &&- 5x_2 &&+ x_4\\ x_3 &= 6 &&- x_1 &&- 4x_2 &&+ x_4. \end{aligned}$$

A solução básica associada é $(x_0, x_1, x_2, x_3, x_4) = (4, 0, 0, 6, 0)$, que é viável. Agora chamamos Pivot repetidamente até obtermos uma solução ótima para L_{aux}. Nesse caso, uma chamada a Pivot com x_2 entrando e x_0 saindo produz

$$\begin{aligned} z &= - x_0 \\ x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5}. \end{aligned}$$

Essa forma de folgas é a solução final para o problema auxiliar. Visto que essa solução tem $x_0 = 0$, sabemos que nosso problema inicial era viável. Além disso, visto que $x_0 = 0$, podemos simplesmente removê-la do conjunto de restrições. Então restauramos a função objetivo original, com substituições apropriadas feitas para incluir somente variáveis não básicas. Em nosso exemplo, obtemos a função objetivo

$$2x_1 - x_2 = 2x_1 - \left(\frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5}\right).$$

Definindo $x_0 = 0$ e simplificando, obtemos a função objetivo

$$-\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5},$$

e a forma de folgas

$$\begin{aligned} z &= -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ x_2 &= \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \end{aligned}$$

Essa forma de folgas tem uma solução básica viável, e podemos retorná-la ao procedimento Simplex.

Agora, mostramos formalmente a correção de Initialize-Simplex.

Lema 29.12

Se um programa linear L não tem nenhuma solução viável, então Initialize-Simplex retorna "inviável". Caso contrário, retorna uma forma de folgas válida para a qual a solução básica é viável.

Prova Primeiro suponha que o programa linear L não tenha nenhuma solução viável. Então, pelo Lema 29.11, o valor objetivo ótimo de L_{aux}, definido em (29.106)–(29.108) é não nulo e, pela restrição de não negatividade para x_0, o valor ótimo objetivo deve ser negativo. Além disso, esse valor objetivo deve ser finito, já que definir $x_i = 0$, para $i = 1, 2, ..., n$ e $x_0 = \left|\min_{i=1}^{m}\{b_i\}\right|$ é viável, e essa solução tem valor objetivo $-\left|\min_{i=1}^{m}\{b_i\}\right|$. Portanto, a linha 10 de INITIALIZE-SIMPLEX encontra uma solução com um valor objetivo não positivo. Seja $\bar{x}$ a solução básica associada à forma de folgas final. Não podemos ter $\bar{x}_0 = 0$, porque L_{aux} teria valor objetivo 0, o que contradiz que o valor objetivo é negativo. Assim, o teste na linha 11 resulta no retorno de "inviável" na linha 16.

Agora, suponha que o programa linear L tenha uma solução viável. Pelo Exercício 29.3-4, sabemos que, se $b_i \geq 0$ para $i = 1, 2, ..., m$, então a solução básica associada à forma de folgas inicial é viável. Nesse caso, as linhas 2-3 retornam a forma de folgas associada à entrada. (Converter a forma-padrão para a forma de folgas é fácil, já que A, b e c são as mesmas em ambas.)

No restante da prova, tratamos o caso no qual o programa linear é viável, mas não retornamos na linha 3. Demonstramos que, nesse caso, as linhas 4-10 encontram uma solução viável para L_{aux} com valor objetivo 0. Primeiro, pelas linhas 1-2, devemos ter

$$b_k < 0$$

e

$$b_k \leq b_i \text{ para cada } i \in B\,. \tag{29.112}$$

Na linha 8, executamos uma operação de pivô na qual a variável de saída x_l (lembre-se de que $l = n = k$, de modo que $b_l < 0$) é o lado esquerdo da equação com b_i mínimo, e a variável de entrada é x_0, a variável extra adicionada. Agora mostramos que, depois desse pivô, todas as entradas de b são não negativas e, consequentemente, a solução básica para L_{aux} é viável. Fazendo x a solução básica depois da chamada a PIVOT e sendo $\hat{b}$ e $\hat{B}$ valores retornados por PIVOT, o Lema 29.1 implica que

$$\bar{x}_i = \begin{cases} b_i - a_{ie}\hat{b}_e & \text{se } i \in \hat{B} - \{e\} \\ b_l / a_{le} & \text{se } i = e \end{cases} \tag{29.113}$$

A chamada a PIVOT na linha 8 tem $e = 0$. Se reescrevermos as desigualdades (29.107) para incluir coeficientes a_{i0},

$$\sum_{j=0}^{n} a_{ij} x_j \leq b_i \quad \text{para } i = 1, 2, ..., m \tag{29.114}$$

então.

$$a_{i0} = a_{ie} = -1 \text{ para cada } i \in B\,. \tag{29.115}$$

(Observe que a_{i0} é o coeficiente de x_0 tal como ele aparece em (29.114), e não a negação do coeficiente porque L_{aux} está na forma-padrão em vez de estar na forma de folgas.) Visto que $l \in B$, também temos que $a_{le} = -1$. Assim, $b_l/a_{le} > 0$ e, portanto, $x_e > 0$. Para as variáveis básicas restantes, temos

$$\begin{aligned} \bar{x}_i &= b_i - a_{ie}\hat{b}_e && \text{(pela equação (29.113))} \\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(pela linha 3 de PIVOT)} \\ &= b_i - b_l && \text{(pela equação (29.115) e } a_{le} = -1) \\ &\geq 0 && \text{(pela desigualdade (29.112))}\,, \end{aligned}$$

que implica que cada variável básica é agora não negativa. Consequentemente, a solução básica depois da chamada a PIVOT na linha 8 é viável. Em seguida, executamos a linha 10, o que resolve L_{aux}. Visto que consideramos que L tem uma solução viável, o Lema 29.11 implica que L_{aux} tem uma solução ótima com valor objetivo 0. Considerando que todas as formas de folga são equivalentes, a solução básica final para L_{aux} deve ter $x_0 = 0$ e, depois de eliminar x_0 do programa linear, obtemos uma forma de folgas que é viável para L. Então, a linha 15 retorna essa forma de folgas.

Teorema fundamental de programação linear

Concluímos este capítulo mostrando que o procedimento SIMPLEX funciona. Em particular, qualquer programa linear é inviável, ilimitado ou tem uma solução ótima com um valor objetivo finito. Em cada caso, SIMPLEX age adequadamente.

Teorema 29.13 (Teorema fundamental de programação linear)

Qualquer programa linear L, dado na forma-padrão,

1. tem uma solução ótima com um valor objetivo finito,
2. é inviável
3. ou é ilimitado.

Se L é inviável, SIMPLEX retorna "inviável". Se L é ilimitado, SIMPLEX retorna "ilimitado". Caso contrário, SIMPLEX retorna uma solução ótima com um valor objetivo finito.

Prova Pelo Lema 29.12, se o programa linear L é inviável, SIMPLEX retorna "inviável". Agora, suponha que o programa linear L seja viável. Pelo Lema 29.12, INITIALIZE-SIMPLEX retorna uma forma de folgas para a qual a solução básica é viável. Portanto, pelo Lema 29.7, SIMPLEX retorna "ilimitado" ou termina com uma solução viável. Se terminar com uma solução finita, o Teorema 29.10 nos diz que essa solução é ótima. Por outro lado, se SIMPLEX retorna "ilimitado", o Lema 29.2 nos diz que o programa linear L é realmente ilimitado. Visto que SIMPLEX sempre termina de um desses modos, a prova está completa.

Exercícios

29.5-1 Dê pseudocódigo detalhado para implementar as linhas 5 e 14 de INITIALIZE-SIMPLEX.

29.5-2 Mostre que, quando o laço principal de SIMPLEX é executado por INITIALIZE-SIMPLEX, o procedimento nunca pode retornar "ilimitado".

29.5-3 Suponha que temos um programa linear L na forma-padrão e que para L e o dual de L as soluções básicas associadas às formas de folga iniciais sejam viáveis. Mostre que o valor objetivo ótimo de L é 0.

29.5-4 Suponha que permitimos desigualdades estritas em um programa linear. Mostre que, nesse caso, o teorema fundamental de programação linear não é válido.

29.5-5 Resolva o seguinte programa linear usando SIMPLEX:

$$\begin{array}{lrcrcr}
\text{maximizar} & x_1 & + & 3x_2 & & \\
\text{sujeito a} & & & & & \\
& x_1 & - & x_2 & \leq & 8 \\
& -x_1 & - & x_2 & \leq & -3 \\
& -x_1 & + & 4x_2 & \leq & 2 \\
& x_1, x_2 & & & \geq & 0.
\end{array}$$

29.5-6 Resolva o seguinte programa linear usando SIMPLEX:

$$\begin{array}{lrcrcr}
\text{maximizar} & x_1 & - & 2x_2 & & \\
\text{sujeito a} & & & & & \\
& x_1 & + & 2x_2 & \leq & 4 \\
& -2x_1 & - & 6x_2 & \leq & -12 \\
& & & x_2 & \leq & 1 \\
& x_1, x_2 & & & \geq & 0.
\end{array}$$

29.5-7 Resolva o seguinte programa linear usando SIMPLEX:

$$\begin{array}{lrcrcr}
\text{maximizar} & x_1 & + & 3x_2 & & \\
\text{sujeito a} & & & & & \\
& -x_1 & + & x_2 & \leq & -1 \\
& -x_1 & - & x_2 & \leq & -3 \\
& -x_1 & + & 4x_2 & \leq & 2 \\
& x_1, x_2 & & & \geq & 0.
\end{array}$$

29.5-8 Resolva o programa linear dado em (29.6)-(29.10).

29.5-9 Considere o seguinte programa linear de uma variável que denominamos P:

$$\begin{array}{lrcl}
\text{maximizar} & tx & & \\
\text{sujeito a} & & & \\
& rx & \leq & s \\
& x & \geq & 0,
\end{array}$$

onde r, s e t são números reais arbitrários. Seja D o dual de P.

Diga para quais valores de r, s e t você pode afirmar que

1. P e D têm soluções ótimas com valores objetivos finitos.
2. P é viável, mas D é inviável.
3. D é viável, mas P é inviável.
4. Nem P nem D é viável.

Problemas

29-1 Viabilidade de desigualdade linear

Dado um conjunto de m desigualdades lineares com n variáveis $x_1, x_2, ..., x_n$, o ***problema de viabilidade de desigualdades lineares*** pergunta se existe uma configuração das variáveis que satisfaça simultaneamente cada uma das desigualdades.

a. Mostre que, se tivermos um algoritmo para programação linear, podemos usá-lo para resolver o problema de viabilidade de desigualdades lineares. O número de variáveis e restrições que você usar no problema de programação linear deve ser polinomial em n e m.

b. Mostre que, se tivermos um algoritmo para problema de viabilidade de desigualdades lineares, podemos usá-lo para resolver um problema de programação linear. O número de variáveis e desigualdades lineares que você usar no problema de viabilidade de desigualdades lineares deve ser polinomial em n e m, número de variáveis e restrições no programa linear.

29-2 Folgas complementares

Folgas complementares descreve uma relação entre os valores de variáveis primais e restrições duais e entre os valores de variáveis duais e restrições primais. Seja x uma solução ótima para o programa linear primal dado em (29.16)-(29.18) e seja y uma solução viável para o programa linear dual dado em (29.83)-(29.85). As folgas complementares dizem que as seguintes condições são necessárias e suficientes para x e y serem ótimas:

$$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \text{ ou } \bar{x}_j = 0 \qquad \text{para } j = 1,2,...,n$$

e

$$\sum_{i=1}^{m} a_{ij}\bar{x}_i = c_j \text{ ou } \bar{y}_j = 0 \qquad \text{para } i = 1,2,...,m\,.$$

a. Verifique se as folgas complementares são válidas para o programa linear nas linhas (29.53)-(29.57).

b. Prove que as folgas complementares são válidas para qualquer programa linear primal e seu dual correspondente.

c. Prove que uma solução viável para um programa linear primal dado nas linhas (29.16)–(29.18) é ótima se e somente se existem valores $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ tais que

1. $\bar{y}$ é uma solução viável para o programa linear dual dado em (29.83)–(29.85),
2. $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$ para todo j tal que $\bar{x}_j > 0$ e
3. $\bar{y}_i = 0$ para todo i tal que $\sum_{i=1}^{m} a_{ij}\bar{x}_j < b_i$.

29-3 Programação linear inteira

Um ***problema de programação linear*** inteira é um problema de programação linear que tem a seguinte restrição adicional: as variáveis x devem ter valores inteiros. O Exercício 34.5-3 mostra que apenas determinar se um programa linear inteiro tem uma solução viável já é NP-difícil, o que significa que não há um algoritmo de tempo polinomial conhecido para esse problema.

a. Mostre que a dualidade fraca (Lema 29.8) é válida para um programa linear inteiro.

b. Mostre que a dualidade (Teorema 29.10) nem sempre é válida para um programa linear inteiro.

c. Dado um programa linear primal na forma-padrão, vamos definir P como o valor objetivo ótimo para o programa linear primal, D como o valor objetivo ótimo para seu dual, IP como o valor objetivo ótimo para a versão inteira do primal (isto é, o primal com a restrição adicional de as variáveis terem valores inteiros) e ID como o valor objetivo ótimo para a versão com inteiros do dual. Mostre que

$IP \leq P = D \leq ID$.

29-4 Lema de Farkas

Seja A uma matriz $m \times n$ e c um n– vetor. Então, o lema de Farkas afirma que exatamente um dos sistemas

$$IP \leq P = D \leq ID.$$

é solúvel, onde x é um n-vetor e y é um m–vetor. Prove o lema de Farkas.

29-5 Circulação de custo mínimo

Neste problema, consideramos uma variante do problema de fluxo de custo mínimo da Seção 29.2 no qual não temos nem uma demanda nem uma fonte, nem um sorvedouro. Em vez disso temos, como antes, uma rede de fluxo e custos de aresta $a(u, v)$. Um fluxo é viável se satisfaz a restrição de capacidade em todas as arestas e de conservação de fluxo em *todos* os vértices. A meta desse problema é determinar, entre todos os fluxos possíveis, o que tem o menor custo, e o denominamos ***problema da circulação de menor custo.***

a. Formule o problema da circulação de menor custo como um programa linear.

b. Suponha que, para todas as arestas $(u, v) \in E$, tenhamos $a(u, v) > 0$. Caracterize uma solução ótima para o problema da circulação de menor custo.

c. Formule o problema de fluxo máximo como um programa linear para o problema de circulação de custo mínimo. Isto é, dada uma instância de problema de custo máximo $G = (V, E)$ com fonte s, sorvedouro t e capacidades de arestas c, crie um problema de circulação de custo mínimo dando uma rede $G' = (V', E')$ (possivelmente diferente) com capacidades de arestas c' e custos de arestas a' tal que você possa obter

uma solução para o problema de fluxo máximo a partir de uma solução para o problema de circulação de custo mínimo.

d. Formule o problema de caminho mínimo de fonte única como um um problema de circulação de custo mínimo.

NOTAS DO CAPÍTULO

Este capítulo só inicia o estudo do amplo campo da programação linear. Vários livros se dedicam exclusivamente à programação linear, inclusive os de Chvátal [69], Gass [130], Karloff [197], Schrijver [303] e Vanderbei [344]. Muitos outros livros apresentam uma ampla abordagem da programação linear, entre eles os de Papadimitriou e Steiglitz [271] e Ahuja, Magnanti e Orlin [7]. A abordagem deste capítulo se baseia na adotada por Chvátal.

O algoritmo simplex para programação linear foi inventado por G. Dantzig em 1947. Logo depois, pesquisadores descobriram como formular vários problemas em diversas áreas como programas lineares e a resolvê-los com o algoritmo simplex. O resultado é que as aplicações da programação linear floresceram, assim como vários algoritmos. Variantes do algoritmo simplex continuam a ser os métodos mais populares para resolver problemas de programação linear. Essa história aparece em vários lugares, inclusive nas notas em [69] e [197].

O algoritmo dos elipsoides foi o primeiro algoritmo de tempo polinomial para programação linear e se deve a L. G. Khachian em 1979; ele se baseou em trabalho anterior de N. Z. Shor, D. B. Judin e A. S. Nemirovskii. Grötschel, Lovász e Schrijver [154] descrevem como usar o algoritmo dos elipsoides para resolver uma variedade de problemas de otimização combinatória. Até agora, parece que o algoritmo dos elipsoides não compete com o algoritmo simplex na prática. O artigo de Karmarkar [198] inclui uma descrição do primeiro algoritmo de pontos interiores. Muitos pesquisadores depois dele projetaram algoritmos de pontos interiores. Boas resenhas aparecem no artigo de Goldfarb e Todd [141] e no livro de Ye [361].

A análise do algoritmo simplex continua sendo uma área ativa de pesquisa. V. Klee e G. J. Minty construíram um exemplo no qual o algoritmo simplex executa 2_n - 1 iterações. O algoritmo simplex, normalmente funciona muito bem na prática, e muitos pesquisadores tentaram dar justificativa teórica para essa observação empírica. Uma linha de pesquisa iniciada por K. H. Borgwardt e seguida por muitos outros mostra que, sob certas hipóteses probabilísticas em relação à entrada, o algoritmo simplex converge em tempo polinomial esperado. Spielman e Teng [322] fizeram progresso nessa área apresentando a "análise suavizada de algoritmos" e aplicando-a ao algoritmo simplex.

O algoritmo simplex é conhecido por funcionar eficientemente em certos casos especiais. Particularmente digno de nota é o algoritmo simplex de rede, que é o algoritmo simplex especializado para problemas de rede de fluxo. Para certos problemas de rede, inclusive os problemas de caminhos mínimos, fluxo máximo e fluxo de custo mínimo, variantes do algoritmo simplex em rede são executadas em tempo polinomial. Consulte, por exemplo, o artigo de Orlin [268] e as citações ali contidas.

---

1 Uma definição intuitiva de região convexa é que ela cumpre o seguinte requisito: para quaisquer dois pontos na região, todos os pontos sobre um segmento de reta entre eles também estão na região.

*N. do RT: esse uso do termo simplex está em desacordo com toda a literatura, entretanto mantivemos a terminologia dos autores.

30 POLINÔMIOS E A FFT

O método direto para somar dois polinômios de grau n demora o tempo $Q(n)$, mas o método direto para multiplicá-los demora o tempo $Q(n_2)$. Neste capítulo, mostraremos como a transformada rápida de Fourier (FFT - *Fast Fourier Transform*) pode reduzir o tempo para multiplicar polinômios a $Q(n \lg n)$.

O uso mais comum das transformadas de Fourier, e consequentemente da FFT, é no processamento de sinais. Um sinal é dado no ***domínio do tempo*****:** como uma função que mapeia tempo para amplitude. A análise de Fourier nos permite expressar o sinal como uma soma ponderada de senoides defasadas de frequências variáveis. Os pesos e fases associados às frequências caracterizam o sinal no ***domínio da frequência***. Entre as muitas aplicações corriqueiras da FFT estão as técnicas de compressão utilizadas para codificar informações digitais de vídeo e áudio, incluindo arquivos MP3. Vários livros de boa qualidade pesquisam a fundo a rica área de processamento de sinais; as notas do capítulo citam alguns deles.

Polinômios

Um ***polinômio*** na variável x em um corpo algébrico F representa uma função $A(x)$ como uma soma formal:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j .$$

Denominamos os valores $a_0, a_1, ..., a_{n-1}$ ***coeficientes*** do polinômio. Os coeficientes são extraídos de um corpo F, normalmente o conjunto de números complexos. Um polinômio $A(x)$ tem ***grau*** k se este é o maior coeficiente não nulo. Qualquer inteiro estritamente maior que o grau de um polinômio é um ***limite do grau*** desse polinômio. Portanto, o grau de um polinômio de grau limitado por n pode ser qualquer inteiro entre 0 e n - 1, inclusive.

Podemos definir uma variedade de operações com polinômios. No caso da ***adição de polinômios***, se $A(x)$ e $B(x)$ são polinômios de grau limitado por n, sua ***soma*** é um polinômio $C(x)$, também de grau limitado por n, tal que $C(x) = A(x) + B(x)$ para todo x no corpo. Isto é, se

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

e

$$B(x) = \sum_{j=0}^{n-1} b_j x^j \, ,$$

então

$$C(x) = \sum_{j=0}^{n-1} c_j x^j \, ,$$

onde $c_j = a_j + b_j$ para $j = 0, 1, ..., n - 1$. Por exemplo, se tivermos os polinômios $A(x) = 6x + 7x_2 - 10x + 9$ e $B(x) = -2x_3 + 4x - 5$, então $C(x) = 4x_3 + 7x_2 - 6x + 4$.

No caso da ***multiplicação de polinômios***, se $A(x)$ e $B(x)$ são polinômios de grau limitado por n, seu ***produto*** $C(x)$ é um polinômio de grau limitado por $2n - 1$ tal que $C(x) = A(x)B(x)$ para todo x no corpo. É provável que você já tenha multiplicado polinômios multiplicando cada termo em $A(x)$ por cada termo em $B(x)$ e combinando termos com potências iguais. Por exemplo, podemos multiplicar $A(x) = 6x_3 + 7x_2 - 10x + 9$ e $B(x) = -2x_3 + 4x - 5$ da seguinte maneira:

$$\begin{array}{rrrrrrr}
 & & & 6x^3 & +\ 7x^2 & -\ 10x & +\ 9 \\
 & & & -\ 2x^3 & & +\ 4x & -\ 5 \\
\hline
 & & & -\ 30x^3 & -\ 35x^2 & +\ 50x & -\ 45 \\
 & & 24x^4 & +\ 28x^3 & -\ 40x^2 & +\ 36x & \\
-\ 12x^6 & -\ 14x^5 & +\ 20x^4 & -\ 18x^3 & & & \\
\hline
-\ 12x^6 & -\ 14x^5 & +\ 44x^4 & -\ 20x^3 & -\ 75x^2 & +\ 86x & -\ 45
\end{array}$$

Outro modo de expressar o produto $C(x)$ é

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \, , \tag{30.1}$$

onde

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \, . \tag{30.2}$$

Observe que grau(C) = grau(A) + grau(B), implicando que, se A é um polinômio de grau limitado por n_a e B é um polinômio de grau limitado por n_b, então C é um polinômio de grau limitado por $n_a + n_b - 1$. Visto que um polinômio de grau limitado por k é também um polinômio de grau limitado a $k + 1$, normalmente diremos que o produto de polinômios C é um polinômio de grau limitado por $n_a + n_b$.

Esboço do capítulo

A Seção 30.1 apresenta dois modos de representar polinômios: a representação por coeficientes e a representação por pares ponto-valor. Os métodos diretos para multiplicar polinômios — equações (30.1) e (30.2) — demoram o

tempo $Q(n_2)$ quando representamos polinômios na forma de coeficientes, mas apenas o tempo $Q(n)$ quando os representamos na forma de pares ponto-valor. Porém, usando a representação por coeficientes, a multiplicação de polinômios demorará somente o tempo $Q(n \lg n)$ se fizermos a conversão entre as duas representações. Para verificar por que essa abordagem funciona, primeiro temos de estudar raízes complexas da unidade, o que fazemos na Seção 30.2. Em seguida, usamos a FFT e sua inversa, também descrita na Seção 30.2, para efetuar as conversões. A Seção 30.3 mostra como implementar a FFT rapidamente em modelos seriais, bem como paralelos.

Este capítulo usa números complexos extensivamente, e o símbolo i será usado exclusivamente para denotar $\sqrt{-1}$.

30.1 REPRESENTAÇÃO DE POLINÔMIOS

As representações de polinômios por coeficientes e por pares ponto-valor são, em certo sentido, equivalentes; isto é, um polinômio na forma de pares ponto-valor tem uma contraparte exclusiva na forma de coeficientes. Nesta seção, apresentamos as duas representações e mostramos como combiná-las de modo a multiplicar dois polinômios de grau limitado a n no tempo $Q(n \lg n)$.

Representação por coeficientes

Uma ***representação por coeficientes*** de um polinômio $A(x) = \sum_{j=0}^{n-1} a_j x^j$ de grau limitado a n é um vetor de coeficientes $a = (a_0, a_1, ..., a_{n-1})$. Nas equações matriciais deste capítulo, em geral trataremos vetores como vetores coluna.

A representação por coeficientes é conveniente para certas operações com polinômios. Por exemplo, a operação de ***avaliar*** o polinômio $A(x)$ em um determinado ponto x_0 consiste em calcular o valor de $A(x_0)$. Podemos avaliar um polinômio no tempo $Q(n)$ usando a ***regra de Horner***:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + ...+ x_0(a_{n-2} + x_0(a_{n-1})) ...)) .$$

De modo semelhante, somar dois polinômios representados pelos vetores de coeficientes $a = (a_0, a_1, ..., a_{n-1})$ e $b = (b_0, b_1, ..., b_{n-1})$ demora o tempo $Q(n)$: simplesmente produzimos o vetor de coeficientes $c = (c_0, c_1, ..., c_{n-1})$, onde $c_j \; a_j + b_j$ para $j = 0, 1, ..., n - 1$.

Agora, considere multiplicar dois polinômios de grau limitado a n, $A(x)$ e $B(x)$, representados na forma de coeficientes. Se usarmos o método descrito pelas equações (30.1) e (30.2), a multiplicação de polinômios demorará o tempo $Q(n_2)$, já que temos de multiplicar cada coeficiente no vetor a por cada coeficiente no vetor b. A operação de multiplicar polinômios em forma de coeficientes parece ser consideravelmente mais difícil que a de avaliar um polinômio ou somar dois polinômios. O vetor de coeficientes c resultante, dado pela equação (30.2), é também denominado ***convolução*** dos vetores de entrada a e b, denotada por $c = a \otimes b$. Visto que multiplicar polinômios e calcular convoluções são problemas computacionais fundamentais de considerável importância prática, este capítulo se concentra em algoritmos eficientes para essas operações.

Representação por pares ponto-valor

Uma ***representação por pares ponto-valor*** de um polinômio $A(x)$ de grau limitado a n é um conjunto de n ***pares ponto-valor***

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$$

tal que todos os valores x_k são distintos e

$$y_k = A(x_k) \tag{30.3}$$

para $k = 0, 1, ..., n - 1$. Um polinômio tem muitas representações por pares ponto-valor diferentes, já que podemos usar qualquer conjunto de n pontos distintos $x_0, x_1, ..., x_{n-1}$ como base para a representação.

Calcular uma representação por pares ponto-valor para um dado polinômio na forma por coeficientes é em princípio direto, já que tudo o que temos de fazer é selecionar n pontos distintos $x_0, x_1, ..., x_{n-1}$, e então avaliar $A(x_k)$ para $k = 0, 1, ..., n - 1$. Com o método de Horner, avaliar um polinômio em n pontos demora o tempo $Q(n_2)$. Mais adiante veremos que, se escolhermos x_k inteligentemente, podemos acelerar essa operação e conseguir tempo de execução $Q(n \lg n)$.

O inverso da avaliação — determinar a forma por coeficientes de um polinômio partindo de uma representação por pares ponto-valor — é a ***interpolação***. O teorema a seguir mostra que a interpolação é bem definida quando o polinômio interpolador desejado deve ser um polinômio de grau limitado igual ao número dado de pares ponto-valor.

Teorema 30.1 (Unicidade de um polinômio interpolador)

Para qualquer conjunto $\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$ de n pares ponto-valor tal que todos os valores x_k são distintos, existe um polinômio único $A(x)$ de grau limitado por n tal que $y_k = A(x_k)$ para $k = 0, 1, ..., n - 1$.

Prova A prova é baseada na existência da inversa de certa matriz. A equação (30.3) é equivalente à equação matricial

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}. \qquad (30.4)$$

A matriz no lado esquerdo é denotada por $V(x_0, x_1, ..., x_{n-1})$ e é conhecida como matriz de Vandermonde. Pelo Problema D-1, essa matriz tem determinante

$$\prod_{0 \le j < k \le n-1} (x_k - x_j),$$

e, portanto, pelo Teorema D.5, ela é inversível (isto é, não singular) se os x_k são distintos. Assim, podemos resolver para os coeficientes a_j unicamente, dada a representação por pares ponto-valor:

$a = V(x_0, x_1, ..., x_{n-1})^{-1}$ y .

A prova do Teorema 30.1 descreve um algoritmo para interpolação baseado na solução do conjunto (30.4) de equações lineares. Usando os algoritmos de decomposição LU do Capítulo 28, podemos resolver essas equações no tempo $O(n_3)$.

Um algoritmo mais rápido para interpolação de n pontos se baseia na ***fórmula de Lagrange***:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \ne k}(x_k - x_j)}{\prod_{j \ne k}(x_k - x_j)}. \qquad (30.5)$$

Seria interessante verificar que o lado direito da equação (30.5) é um polinômio de grau limitado por n que satisfaz $A(x_k) = y_k$ para todo k. O Exercício 30.1-5 pede que você mostre como calcular os coeficientes de A no tempo $Q(n_2)$ usando a fórmula de Lagrange.

Assim, avaliação e interpolação de n pontos são operações inversas bem definidas que transformam a representação de um polinômio por coeficientes em uma representação por pares ponto-valor e vice-versa. [1] Os algoritmos já descritos para esses problemas demoram o tempo $Q(n_2)$.

A representação por pares ponto-valor é bastante conveniente para muitas operações com polinômios. No caso da adição, se $C(x) = A(x) + B(x)$, então $C(x_k) = A(x_k + B(x_k)$ para qualquer ponto x_k. Mais precisamente, se temos uma representação por pares ponto-valor para A,

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\} ,$$

e, para B,

$$\{(x_0, y'_0), (x_1, x_1), ..., (x_{n-1}, y'_{n-1})\}$$

(observe que A e B são avaliados nos *mesmos n* pontos), então uma representação por pares ponto-valor para C é

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), ..., (x_{n-1}, y_{n-1} + y'_{n-1})\} .$$

Assim, o tempo para somar dois polinômios de grau limitado a n na forma com pares ponto-valor é $Q(n)$.

De modo semelhante, a representação por pares ponto-valor é conveniente para multiplicar polinômios. Se $C(x) = A(x)B(x)$, então $C(x_k) = A(x_k)B(x_k)$ para qualquer ponto x_k, e podemos multiplicar ponto a ponto uma representação por pares ponto-valor para A por uma representação por pares ponto-valor para B para obter uma representação por pares ponto-valor para C. Contudo, temos de encarar o seguinte problema: grau (C) = grau(A) + grau(B); se A e B são polinômios de grau limitado por n, então C é um polinômios de grau limitado por $2n$. Uma representação-padrão por pares ponto-valor para A e B consiste em n pares ponto-valor para cada polinômio. Quando nós os multiplicamos, obtemos n pares ponto-valor, mas precisamos de $2n$ pares para interpolar um polinômio único c de grau limitado por $2n$ (veja o Exercício 30.1-4). Portanto, devemos começar com representações por pares ponto-valor "estendidas" para A e B, consistindo em $2n$ pares ponto-valor cada uma. Dada uma representação por pares ponto-valor estendida para A,

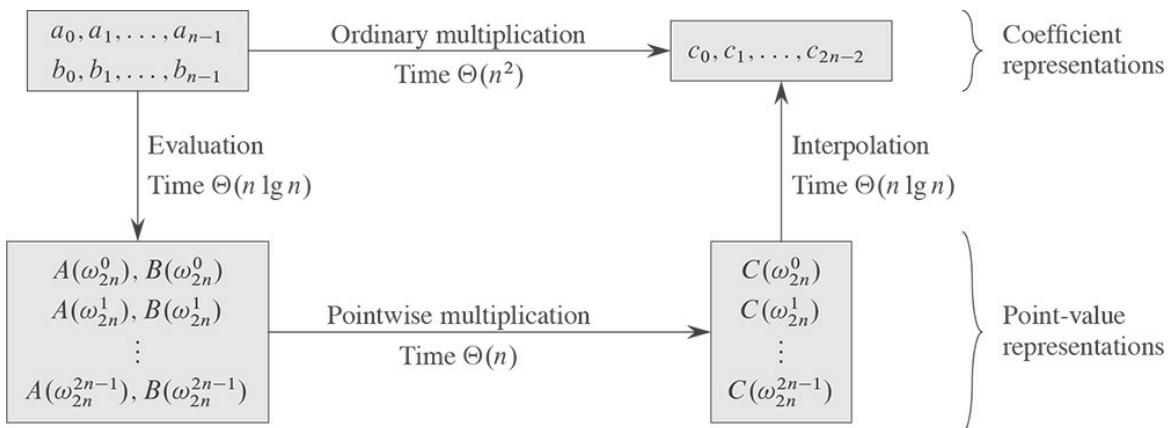
$$\{(x_0, y_0), (x_1, y_1), ..., (x_{2n-1}, y_{2n-1})\} ,$$

e uma representação por pares ponto-valor estendida correspondente para B,

$$\{(x_0, y'_0), (x_1, y'_1), ..., (x_{2n-1}, y'_{2n-1})\} ,$$

então uma representação por pares ponto-valor para C é

$$\{(x_0, y_0, y'_0), (x_1, y_1, y'_1), ..., (x_{2n-1}, y_{2n-1}, y'_{2n-1})\} ,$$

Dados dois polinômios de entrada na forma com pares ponto-valor estendida, vemos que o tempo para multiplicá-los de modo a obter a forma com pares ponto-valor do resultado é $Q(n)$, muito menor que o tempo requerido para multiplicar polinômios na forma com coeficientes.

Finalmente, consideramos como avaliar um polinômio dado na forma com pares ponto-valor em um novo ponto. Para esse problema, não conhecemos nenhuma abordagem mais simples que converter o polinômio primeiro para a forma com coeficientes e depois avaliá-lo no novo ponto.

Multiplicação rápida de polinômios na forma com coeficientes

Podemos usar o método de multiplicação em tempo linear para polinômios na forma com pares ponto-valor para acelerar a multiplicação de polinômios na forma com coeficientes? A resposta depende de podermos ou não converter um polinômio rapidamente da forma com coeficientes para a forma com pares ponto-valor (avaliar) e vice-versa (interpolar).

Podemos usar quaisquer pontos que quisermos como pontos de avaliação mas, escolhendo os pontos de avaliação cuidadosamente, a conversão entre representações demora apenas o tempo $Q(n \lg n)$. Como veremos na Seção 30.2, se escolhermos "raízes complexas da unidade" como pontos de avaliação, poderemos produzir uma representação por pares ponto-valor tomando a transformada discreta de Fourier (DFT — *Discrete Fourier Transform*) de um vetor de coeficientes. Podemos efetuar a operação inversa, a interpolação, tomando a "DFT inversa" de pares ponto-valor, produzindo um vetor de coeficientes. A Seção 30.2 mostrará como a FFT executa as operações DFT e DFT inversa no tempo $Q(n \lg n)$.

A Figura 30.1 mostra essa estratégia em gráfico. Um pequeno detalhe referente às limitações de grau: o produto de dois polinômios de grau limitado por n é um polinômio de grau limitado por $2n$. Portanto, antes de avaliar os polinômios de entrada A e B, primeiro temos de dobrar as limitações de grau para $2n$ somando n coeficientes de ordem alta iguais a 0. Como os vetores têm $2n$ elementos, usamos "raízes $(2n)$-ésimas complexas da unidade", que são denotadas pelos $^{2}{}_{n}$ termos na Figura 30.1.

Figura 30.1 Esboço gráfico de um processo eficiente de multiplicação de polinômios. As representações na parte superior estão na forma com coeficientes, enquanto as da parte inferior estão na forma com pares ponto-valor. As setas da esquerda para a direita correspondem à operação de multiplicação. Os $_{2n}$ termos são raízes complexas $(2n)$-ésimas da unidade.

Dada a FFT, temos o procedimento de tempo $Q(n \lg n)$ descrito a seguir para multiplicar dois polinômios $A(x)$ e $B(x)$ de grau limitado a n, onde as representações de entrada e saída estão na forma com coeficientes. Supomos que n é uma potência de 2; sempre podemos cumprir esse requisito adicionando coeficientes de ordem alta iguais a zero.

1. *Dobrar a limitação do grau*: Crie representações por coeficientes de $A(x)$ e $B(x)$ como polinômios de grau limitado por $2n$, adicionando n coeficientes zero de ordem alta a cada um.
2. *Avaliar*: Calcule representações por pares ponto-valor de $A(x)$ e $B(x)$ de comprimento $2n$ aplicando a FFT de ordem $2n$ a cada polinômio. Essas representações contêm os valores dos dois polinômios nas raízes $(2n)$-ésimas da unidade.
3. *Multiplicação ponto a ponto*: Calcule uma representação por pares ponto-valor para o polinômio $C(x) = A(x)B(x)$ multiplicando esses valores ponto a ponto. Essa representação contém o valor de $C(x)$ em cada raiz $(2n)$-ésima da unidade.
4. *Interpolar*: Crie a representação por coeficientes do polinômio $C(x)$ aplicando a FFT a $2n$ pares ponto-valor para calcular a DFT inversa.

As etapas (1) e (3) demoram o tempo $Q(n)$, e as etapas (2) e (4) demoram o tempo $Q(n \lg n)$. Assim, uma vez demonstrado como usar a FFT, teremos provado o seguinte.

Teorema 30.2

Podemos multiplicar dois polinômios de grau limitado a n no tempo $Q(n \lg n)$, com ambas as representações de entrada e saída na forma com coeficientes.

Exercícios

30.1-1 Multiplique os polinômios $A(x) = 7x_3 - x_2 + x - 10$ e $B(x) = 8x_3 - 6x + 3$, usando as equações (30.1) e (30.2).

30.1-2 Um outro modo de avaliar um polinômio $A(x)$ de grau limitado a n em um dado ponto x_0 é dividir $A(x)$ pelo polinômio $(x - x_0)$, obtendo um polinômio quociente $q(x)$ de grau limitado a $n - 1$ e um resto r, tais que

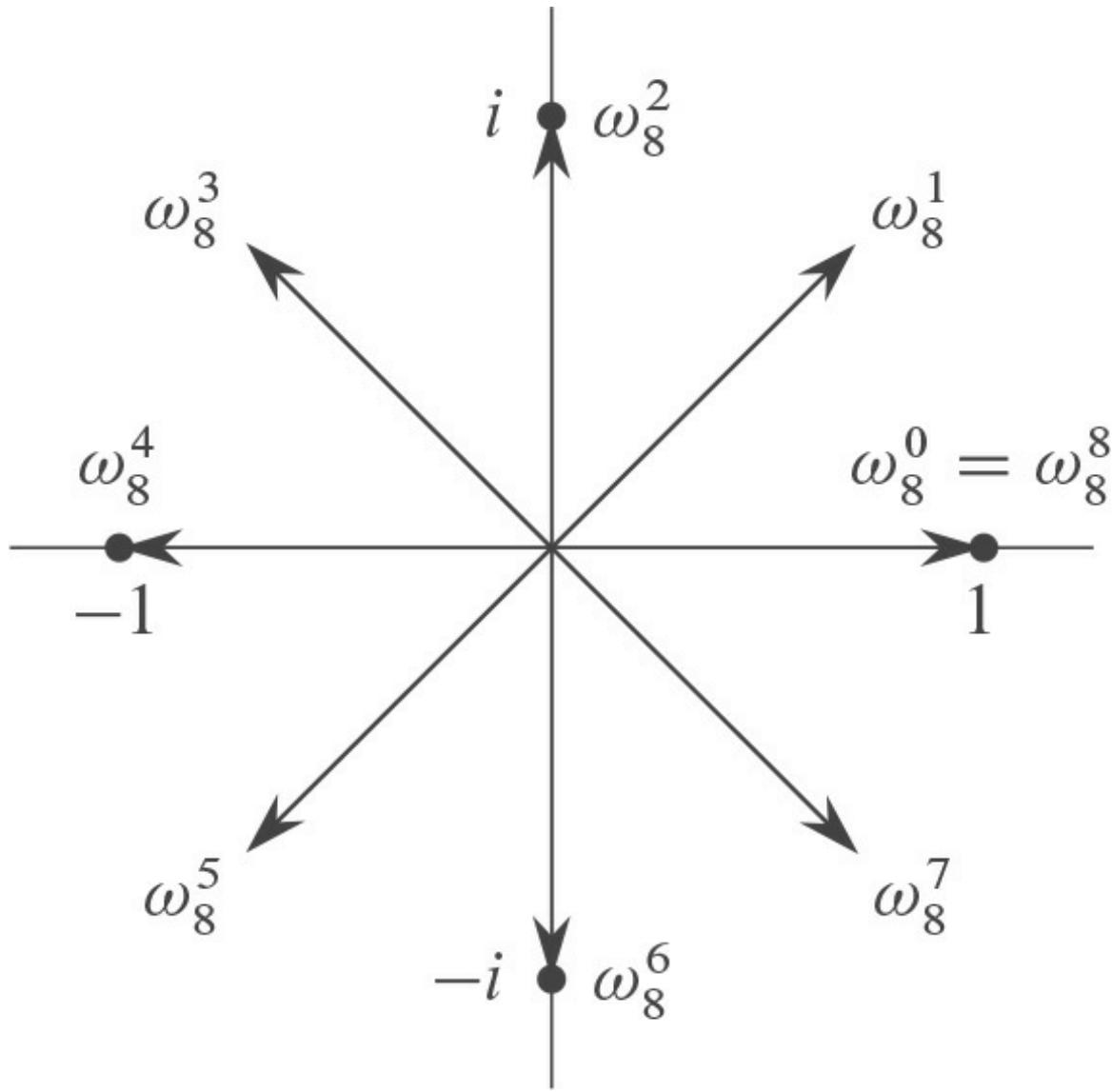
$$A(x) = q(x)(x - x_0) + r\ .$$

Claramente, $A(x_0) = r$. Mostre como calcular o resto r e os coeficientes de $q(x)$ no tempo $Q(n)$ dados x_0 e os coeficientes de A.

30.1-3 Deduza uma representação por pares ponto-valor para $A_{\text{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$ partindo de uma representação por pares ponto-valor para $A(x) = \sum_{j=0}^{n-1} a_j x^j$, supondo que nenhum dos pontos é 0.

30.1-4 Prove que são necessários n pares ponto-valor distintos para especificar unicamente um polinômio de grau limitado por n, isto é, se são dados menos do que n pares ponto-valor distintos, eles não conseguem especificar um polinômio de grau limitado por n único. (*Sugestão:* Usando o Teorema 30.1, o que você pode dizer sobre um conjunto de $n - 1$ pares ponto-valor ao qual acrescenta um par ponto-valor escolhido de modo arbitrário?)

30.1-5 Mostre como usar a equação (30.5) para interpolar no tempo $Q(n_2)$. (*Sugestão*: Primeiro calcule a representação por coeficientes do polinômio $\prod j\ (x - x_j)$ e então divida por $(x - x_j)$ conforme necessário para o numerador de cada termo. Veja o Exercício 30.1-2. Podemos calcular cada um dos n denominadores no tempo $O(n)$.)

30.1-6 Explique o que está errado na abordagem "óbvia" para divisão de polinômios usando uma representação por pares ponto-valor, isto é, dividindo os valores y correspondentes. Discuta separadamente o caso em que o resultado da divisão é exato e o caso em que não é exato.

30.1-7 Considere dois conjuntos A e B, cada um com n inteiros na faixa de 0 a $10n$. Desejamos calcular a ***soma cartesiana*** de A e B, definida por

$$C = \{x + y : x \in A \text{ e } y \in B\}.$$

Observe que os inteiros em C estão na faixa de 0 a $20n$. Queremos determinar os elementos de C e o número de vezes que cada elemento de C é obtido como uma soma de elementos em A e B. Mostre como resolver o problema no tempo $O(n \lg n)$. (*Sugestão*: Represente A e B como polinômios de grau no máximo $10n$.)

30.2 DFT E FFT

Na Seção 30.1, afirmamos que, se usarmos raízes complexas da unidade, podemos avaliar e interpolar polinômios no tempo Q(n lg n). Nesta seção, definimos raízes complexas da unidade e estudamos suas propriedades, definimos a DFT e depois mostramos como a FFT calcula a DFT e sua inversa no tempo Q(n lg n).

Figura 30.2 Os valores de ω^0_8 ,ω^1_8 , ,ω^7_8 no plano complexo, onde $w_8 = e^{2\pi i/8}$ é a raiz oitava principal da unidade.

Raízes complexas da unidade

Uma ***raiz complexa n-ésima da unidade*** é um número complexo tal que

$$\omega^n = 1 \, .$$

Há exatamente n raízes n-ésimas complexas da unidade: $e_{2p}{}^{ik}/_{n}$ para k = 0, 1, ..., n - 1. Para interpretar essa fórmula, usamos a definição da exponencial de um número complexo:

$$e^{iu} = \cos(u) + i\,\text{sen}(u)\ .$$

A Figura 30.2 mostra que as n raízes complexas da unidade estão igualmente espaçadas ao redor do círculo de raio unitário com centro na origem do plano complexo. O valor

$$\omega_n = e^{2\pi i/n} \tag{30.6}$$

é ***raiz n-ésima principal da unidade***;2 todas as outras raízes n-ésimas complexas da unidade são potências de $_n$.

As n raízes n-ésimas complexas da unidade,

$$\omega_n^0, \omega_n^1, \cdots, \omega_n^{n-1},$$

formam um grupo sob multiplicação (consulte a Seção 31.3). Esse grupo tem a mesma estrutura que o grupo aditivo $(\mathbb{Z}_{n,} +)$ módulo n, já $\omega_n^n = \omega_n^0 = 1$ que implica que $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. De modo semelhante, $\omega_n^{-1} = \omega_n^{n-1}$. Os lemas apresentados a seguir dão algumas propriedades essenciais das raízes n-ésimas complexas da unidade.

Lema 30.3 (Lema do cancelamento)

Para quaisquer inteiros $n \geq 0$, $k \geq 0$ e $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k. \tag{30.7}$$

Prova O lema decorre diretamente da equação (30.6), visto que

$$\omega_{dn}^{dk} = (e^{2\pi i/dn})^{dk}$$
$$= (e^{2\pi i/n})^k$$
$$= \omega_n^k.$$

Corolário 30.4

Para qualquer inteiro par $n > 0$,

$$\omega_n^{n/2} = \omega_2 = -1$$

Prova A prova fica para o Exercício 30.2-1.

Lema 30.5 (Lema da divisão em metades)

Se $n > 0$ é par, então os quadrados das n raízes n-ésimas complexas da unidade são as $n/2$ raízes $(n/2)$-ésimas complexas da unidade.

Prova Pelo lema do cancelamento, temos $({}^k_n)^2 = {}^{kn/2}$, para qualquer inteiro não negativo k. Observe que, se elevarmos ao quadrado todas as raízes n-ésimas complexas da unidade, cada >raiz $(n/2)$-ésima da unidade será obtida exatamente

duas vezes, visto que

$$\begin{aligned}(\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k}\omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2.\end{aligned}$$

Assim, ω_n^k e $\omega_n^{k+n/2}$ têm o mesmo quadrado. Também poderíamos ter usado o Corolário 30.4 para provar essa propriedade, já que $\omega_n^{n/2} = -1$ implica $\omega_n^{k+n/2} = -\omega_n^k$ e, portanto $(\omega_n^{k+n/2})^2 = -(\omega_n^k)^2$. .

Como veremos, o lema da divisão em metades é essencial para nossa abordagem de divisão e conquista para converter representações por coeficientes em representações por pares ponto-valor e vice-versa, já que garante que os subproblemas recursivos terão somente metade do tamanho.

Lema 30.6 (Lema do somatório)

Para qualquer inteiro $n \geq 1$ e inteiro não nulo k não divisível por n,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

Prova A equação (A.5) se aplica a valores complexos, bem como a reais, e então temos

$$\begin{aligned}\sum_{j=0}^{n-1} (\omega_n^k)^0 &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\ &= \frac{(\omega_n^n)^k - 1}{\omega^{\frac{k}{n}} - 1} \\ &= \frac{(1)^k - 1}{\omega_n^k - 1} \\ &= 0.\end{aligned}$$

Como impomos que k não seja divisível por n, e como $^k = 1$ somente quando k é divisível por n, garantimos que o denominador não é 0.

A DFT

Lembre-se de que desejamos avaliar um polinômio

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

de grau limitado por n em $\omega_n^0, \omega_n^1, \omega_n^2, \cdots, \omega_n^{n-1}$ (isto é, nas n raízes n-ésimas complexas da unidade).[3] Supomos que A seja dado na forma com coeficientes: $a = (a_0, a_1, ..., a_{n-1})$. Vamos definir os resultados y_k, para $k = 0, 1, ..., n - 1$, por

$$\begin{aligned} y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj}. \end{aligned} \tag{30.8}$$

O vetor $y = (y_0, y_1, ..., y_{n-1})$ é a ***transformada discreta de Fourier*** (***DFT — Discrete Fourier Trans- form***) do vetor de coeficientes $a = (a_0, a_1, ..., a_{n-1})$. Escrevemos também $y = \text{DFT}_n(a)$.

A FFT

Usando um método conhecido como ***transformada rápida de Fourier*** (***FFT — Fast Fourier Transform***), que aproveita as propriedades especiais das raízes complexas da unidade, podemos calcular $\text{DFT}_n(a)$ no tempo Q(n lg n), em comparação com o tempo Q(n_2) do método direto. Supomos em tudo que n é uma potência exata de 2. Embora existam estratégias para lidar com tamanhos que não sejam potências de 2, elas estão fora do escopo deste livro.

O método da FFT emprega uma estratégia de divisão e conquista, utilizando separadamente os coeficientes de índice par e os coeficientes de índice ímpar de $A(x)$ para definir os dois novos polinômios de grau limitado por $n/2$ $A_{[0]}(x)$ e $A_{[1]}(x)$:

$$A^{[0]}(x) = a_0 + a_{2x} + a_{4x}{}^2 + ... + a_{n-2x}{}^{n/2-1} ,$$

$$A^{[1]}(x) = a_1 + a_{3x} + a_{5x}{}^2 + ... + a_{n-1x}{}^{n/2-1} .$$

Observe que $A^{[0]}$ contém todos os coeficientes de índice par de A (a representação binária do índice termina em 0) e $A^{[1]}$ contém todos os coeficientes de índice ímpar (a representação binária do índice termina em 1). Decorre que

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) , \tag{30.9}$$

de modo que o problema de avaliar $A(x)$ em $\omega_n^0, \omega_n^1, \cdots, \omega_n^{n-1}$ se reduz a

1. avaliar os polinômios de grau limitado por $n/2$ $A^{[0]}(x)$ e $A^{[1]}(x)$ nos pontos

$$(\omega_n^0)^2, (\omega_n^1)^2, \cdots, (\omega_n^{n-1})^2 \tag{30.10}$$

e então

2. combinar os resultados de acordo com equação (30.9).

Pelo lema da divisão em metades, a lista de valores (30.10) não consiste em n valores distintos, mas somente nas $n/2$ raízes $(n/2)$-ésimas complexas da unidade, sendo que cada raiz ocorre exatamente duas vezes. Portanto, avaliamos recursivamente os polinômios $A_{[0]}$ e $A_{[1]}$ de grau limitado por $n/2$ nas $n/2$ raízes $(n/2)$-ésimas complexas da unidade. Esses subproblemas têm exatamente a mesma forma do problema original, mas metade do tamanho. Agora conseguimos dividir o cálculo de uma DFT_n de n elementos em dois cálculos de $\text{DFT}_{n/2}$ de $n/2$ elementos. Essa

decomposição é a base para o algoritmo recursivo da FFT a seguir, que calcula a DFT de um vetor de n elementos a = $(a_0, a_1, ..., a_{n-1})$, onde n é uma potência de 2.

RECURSIVE-FFT(a)

1 $n = a.comprimento$ // n é uma potência de 2
2 **if** $n == 1$
3 **return** a
4 $\omega_n = e^{2\pi i/n}$
5 $\omega = 1$
6 $a^{[0]} = (a_0, a_2, ..., a_{n-2})$
7 $a^{[1]} = (a_0, a_3, ..., a_{n-1})$
8 $y^{[0]} =$ RECURSIVE-FFT($a^{[0]}$)
9 $y^{[1]} =$ RECURSIVE-FFT($a^{[1]}$)
10 **for** $k = 0$ **to** $n/2 - 1$
11 $y_k = y\,k_k^{[0]} + \omega y_k^{[1]}$
12 $y_{k+(n/2)} = y_k^{[0]} - \omega y_k^{[1]}$
13 $\omega = \omega\,\omega_n$
14 **return** y // y é considerado um vetor coluna

O procedimento Recursive-Fft funciona da seguinte maneira: as linhas 2-3 representam a base da recursão; a DFT de um elemento é o próprio elemento, já que, nesse caso,

$$\begin{aligned} y_0 &= a_0 \omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 \cdot \end{aligned}$$

As linhas 6-7 definem os vetores de coeficientes para os polinômios $A_{[0]}$ e $A_{[1]}$. As linhas 4, 5 e 13 garantem que é atualizado corretamente de modo que, sempre que as linhas 11-12 são executadas, temos $\omega = \omega^k$. (Manter um valor contínuo de iteração a iteração poupa tempo em relação a calcular ω^k desde o início a cada passagem pelo laço **for**.) As linhas 8-9 executam os cálculos recursivos de $DFT_{n/2}$ definindo, para k = 0, 1, ..., n/2 - 1,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k), \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k), \end{aligned}$$

ou, visto que $\omega_{n/2}^k = \omega_n^{2k}$ pelo lema do cancelamento,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_n^{2k}), \\ y_k^{[1]} &= A^{[1]}(\omega_n^{2k}). \end{aligned}$$

As linhas 11−12 combinam os resultados dos cálculos recursivos das $\text{DFT}_{n/2}$. Para $y_0, y_1, ..., y_{n/2-1}$, a linha 11 produz

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) && \text{(pela equação (30.9)).} \end{aligned}$$

Para $y_{n/2}, y_{n/2+1}, ..., y_{n-1}$, fazendo $k = 0, 1, ..., n/2 - 1$, a linha 12 produz

$$\begin{aligned} y_k &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} && \text{(já que } \omega_n^{k+(n/2)} = \omega_n^k) \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2+k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) && \text{(já que } \omega_n^{2k+n} = \omega_n^{2k}) \\ &= A(\omega_n^{k+(n/2)}) && \text{(pela equação 30.9)).} \end{aligned}$$

Assim, o vetor y retornado por RECURSIVE-FFT é de fato a DFT do vetor de entrada a.

As linhas 11 e 12 multiplicam cada valor $y_k^{[1]}$ por ω_n^k, para $k = 0, 1, ..., n/2 - 1$.

A linha 11 soma esse produto a $y_k^{[0]}$, e a linha 12 o subtrai. Como usamos cada fator ω_n^k nas formas positiva e negativa, denominamos os fatores ω_n^k ***fatores de giro***.

Para determinar o tempo de execução do procedimento RECURSIVE-FFT, observamos que, com exceção das chamadas recursivas, cada invocação demora o tempo $\Theta(n)$, onde n é o comprimento do vetor de entrada. Portanto, a recorrência para o tempo de execução é

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) . \end{aligned}$$

Assim, podemos avaliar um polinômio de grau limitado a n nas raízes n-ésimas complexas da unidade no tempo $Q(n \lg n)$ usando a transformada rápida de Fourier.

Interpolação nas raízes complexas da unidade

Agora, completamos o esquema de multiplicação de polinômios mostrando como interpolar as raízes complexas da unidade por um polinômio, o que nos permite reconverter da forma com pares ponto-valor para a forma com coeficientes. Interpolamos escrevendo a DFT como uma equação matricial e depois observando a forma da inversa da matriz.

Pela equação (30.4), podemos escrever a DFT como o produto de matrizes $y = V_n a$, onde V_n é uma matriz de Vandermonde que contém as potências adequadas de ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

A entrada (k, j) de V_n é ω_n^{kj} para $j, k = 0, 1, \ldots n-1$. Os expoentes das entradas de V_n formam uma tabela de multiplicação.

Para a operação inversa, que escrevemos como $a = \mathrm{DFT}_n^{-1}(y)$, multiplicamos y pela matriz V_n^{-1}, a inversa de V_n.

Teorema 30.7

Para $j, k = 0, 1, \ldots n-1$, a entrada (j, k) de é $\sum_{j=0}^{n-1}(\omega_n^k)^0 = \frac{(\omega_n^k)^n - 1}{\omega_-^k - 1}$.

Prova Mostraremos que $V_n^{-1} = I_n$, a matriz identidade $n \times n$. Considere a entrada (j, j') de $V_n^{-1}\,V$:

$$\left[V_n^{-1}\,V_n\right]_{jj'} = \sum_{k=0}^{n-1} ({}_i\omega_n^{-kj}/n)(\omega_n^{kj'})$$
$$\sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n$$

Esse somatório é igual a 1 se $j' = j$, e é 0 caso contrário, de acordo com o lema do somatório (Lema 30.6). Observe que nos baseamos em $-(n-1) < j \le -j' \le n-1$, de modo que $j' - j$ não é divisível por n, o que permite aplicar o lema do somatório. ■

Dada a matriz inversa V_n^{-1}, temos que $\mathrm{DFT}_n^{-1}(y)$ é dada por

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{30.11}$$

para $j = 0, 1, \ldots n-1$. Comparando as equações (30.8) e (30.11), vemos que modificando o algoritmo da FFT para trocar os papéis de a e y, substituir ω_n por ω_n^{-1} e dividir cada elemento do resultado por n, calculamos a DFT inversa (veja o Exercício 30.2-4). Assim, podemos calcular DFT_n^{-1} também no tempo $\Theta(n \lg n)$.

Vemos que, usando a FFT e a FFT inversa, podemos alternar livremente um polinômio de grau limitado por n entre sua representação por coeficientes e sua representação por pares ponto-valor no tempo $\Theta(n \lg n)$. No contexto da multiplicação de polinômios, já mostramos o seguinte.

Teorema 30.8 (Teorema de convolução)

Para quaisquer dois vetores a e b de comprimento n, onde n é uma potência de 2,

$$a \otimes b = \text{DFT}_{2n}^{-1}(\text{DFT}_{2n}(a) \cdot \text{DFT}_{2n}(b)) \ ,$$

onde os vetores a e b são preenchidos com zeros até o comprimento $2n$, e · denota o produto componente a componente de dois vetores de $2n$ elementos.

Exercícios

30.2-1 Prove o Corolário 30.4.

30.2-2 Calcule a DFT do vetor (0, 1, 2, 3).

30.2-3 Faça o Exercício 30.1-1 usando o esquema de tempo $Q(n \lg n)$.

30.2-4 Escreva pseudocódigo para calcular DFT-$^1{}_n$ no tempo $Q(n \lg n)$.

30.2-5 Descreva a generalização do procedimento FFT para o caso em que n é uma potência de 3. Dê uma recorrência para o tempo de execução e resolva a recorrência.

30.2-6 ★ Suponha que, em vez de executar uma FFT de n elementos no corpo dos números complexos (onde n é par), usamos o anel $_m$ de inteiros módulo m, onde $m = 2^{tn/2} + 1$ e t é um inteiro positivo arbitrário. Use $= 2^t$ em vez de $_n$ como uma raiz n-ésima principal da unidade, módulo m. Prove que a DFT e a DFT inversa estão bem definidas nesse sistema.

30.2-7 Dada uma lista de valores $z_0, z_1, ..., z_{n-1}$ (possivelmente com repetições), mostre como determinar os coeficientes de um polinômio $P(x)$ de grau limitado por n que tem zeros somente em $z_0, z_1, ..., z_{n-1}$ (possivelmente com repetições). Seu procedimento deve ser executado no tempo $O(n \lg^2 n)$. (*Sugestão*: O polinômio $P(x)$ tem um zero em z_j se e somente se $P(x)$ é um múltiplo de $(x - z_j)$.)

30.2-8 ★ A ***transformada chirp*** de um vetor a = $(a_0, a_1, ..., a_{n-1})$ é o vetor $y = (y_0, y_1, ..., y_{n-1})$, onde $Y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ é qualquer número complexo. Portanto, a DFT é um caso especial da transformada chirp obtida tomando $z = {}_n$. Mostre como avaliar a transformada chirp no tempo O($n \lg n$) para qualquer número complexo z. (*Sugestão*: Use a equação

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} (a_j z^{j^2/2})(z^{(k-j)^2/2})$$

para ver a transformada chirp como uma convolução.)

30.3 Implementações eficientes de FFT

Considerando que as aplicações práticas da DFT, como o processamento de sinais, exigem a máxima velocidade, esta seção examina duas implementações eficientes de FFT. Primeiro, examinaremos uma versão iterativa do algoritmo da FFT que é executada no tempo $Q(n \lg n)$, mas que pode ter uma constante oculta mais baixa na notação Q que a implementação recursiva da Seção 30.2. (Dependendo da exata implementação, a versão recursiva pode usar a cache

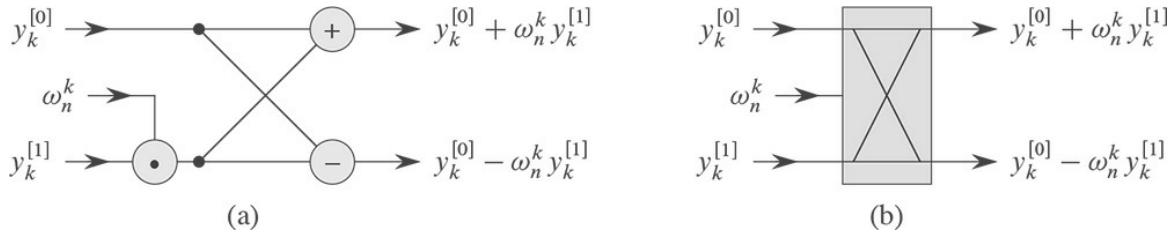
do hardware com maior eficiência.) Então, usaremos as ideias que nos levaram à implementação iterativa para projetar um circuito FFT paralelo eficiente.

Uma implementação iterativa de FFT

Em primeiro lugar observamos que o laço **for** das linhas 10-13 de Recursive-Fft envolve calcular o valor duas vezes. Em terminologia de compilador, esse valor é denominado ***subexpressão comum***. Podemos mudar o laço para calculá-lo apenas uma vez, armazenando-o em uma variável temporária t.

$$
\begin{aligned}
&\textbf{para } k = 0 \textbf{ para } n/2 - 1 \\
&\quad t = \omega y_k^{[1]} \\
&\quad y_k = y_k^{[0]} + t \\
&\quad y_{k+(n/2)} = y_k^{[0]} - t \\
&\quad \omega = \omega \omega_n
\end{aligned}
$$

A operação nesse laço, que é multiplicar o fator de giro $\omega = \omega k_n$ por $y[1]_k$, armazenar o produto em t e adicionar e subtrair t de $y[0]_k$ é conhecida como ***operação borboleta*** e é mostrada esquematicamente na Figura 30.3.

Agora, mostramos como transformar a estrutura do algoritmo FFT em iterativa, em vez de recursiva. Na Figura 30.4, organizamos os vetores de entrada para as chamadas recursivas em uma invocação de Recursive-Fft em uma estrutura de árvore, onde a chamada inicial é para $n = 8$. A árvore tem um nó para cada chamada do procedimento, identificado pelo vetor de entrada correspondente. Cada invocação de Recursive-Fft faz duas chamadas recursivas, a menos que tenha recebido um vetor de um elemento. A primeira chamada aparece no filho à esquerda e a segunda chamada aparece no filho à direita.

Examinando a árvore, observamos que, se pudéssemos organizar os elementos do vetor inicial a na ordem em que eles aparecem nas folhas, poderíamos seguir o curso da execução do procedimento Recursive-fft, mas de baixo para cima, em vez de de cima para baixo. Primeiro, tomamos os elementos aos pares, calculamos a DFT de cada par usando uma operação borboleta e substituímos o par por sua DFT. Então, o vetor contém $n/2$ DFTs de dois elementos. Em seguida, tomamos essas $n/2$ DFTs aos pares e calculamos a DFT dos quatro elementos de vetor de onde elas vieram, executando duas operações borboleta, substituindo duas DFTs de dois elementos por uma DFT de quatro elementos. Então o vetor contém $n/4$ DFTs de quatro elementos. Continuamos dessa maneira até o vetor conter duas DFTs de $(n/2)$ elementos, as quais combinamos usando $n/2$ operações borboleta na DFT final de n elementos.

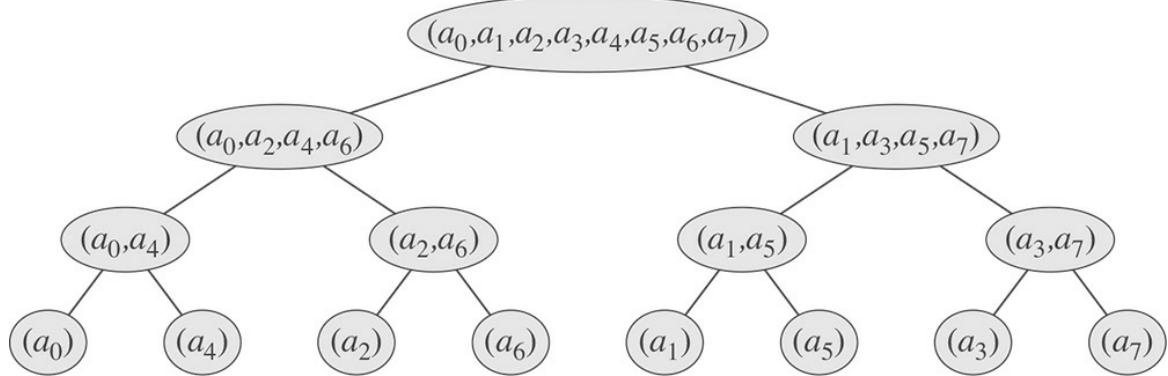
Para transformar essa abordagem de baixo para cima em código, usamos um arranjo $A[0 \,..\, n - 1]$ que inicialmente contém os elementos do vetor de entrada a na ordem em que eles aparecem nas folhas da árvore da Figura 30.4. (Mais adiante, mostraremos como determinar essa ordem, conhecida como permutação com inversão de bits.) Como temos de combinar DFTs em cada nível da árvore, introduzimos uma variável s para contar os níveis, que varia de 1 (na parte inferior, quando estamos combinando pares para formar DFTs de dois elementos) a $\lg n$ (na parte superior, quando estamos combinando duas DFTs de $(n/2)$ elementos para produzir o resultado final.) Portanto, o algoritmo tem a seguinte estrutura:

1 **for** $s = 1$ **to** $\lg n$
2   **for** $k = 0$ **to** $n - 1$ por 2^s
3   combine as duas DFTs de 2^{s-1} elementos em
          $A[k \,..\, k + 2^{s-1} - 1]$ e $A[k + 2^{s-1} \,..\, k + 2^s - 1]$
          em uma DFT de 2^s elementos em $A[k \,..\, k + 2^s - 1]$

Figura 30.3 Uma operação borboleta. (**a**) Os dois valores de entrada entram pela esquerda, o fator de giro $^{k}{}_{n}$ é multiplicado por $y_{k[1]}$, e a soma e a diferença saem pela direita. (**b**) Desenho simplificado de uma operação borboleta. Usaremos essa representação em um circuito FFT paralelo.

Figura 30.4 Árvore de vetores de entrada para as chamadas recursivas do procedimento RECURSIVE-FFT. A invocação inicial é para $n = 8$.

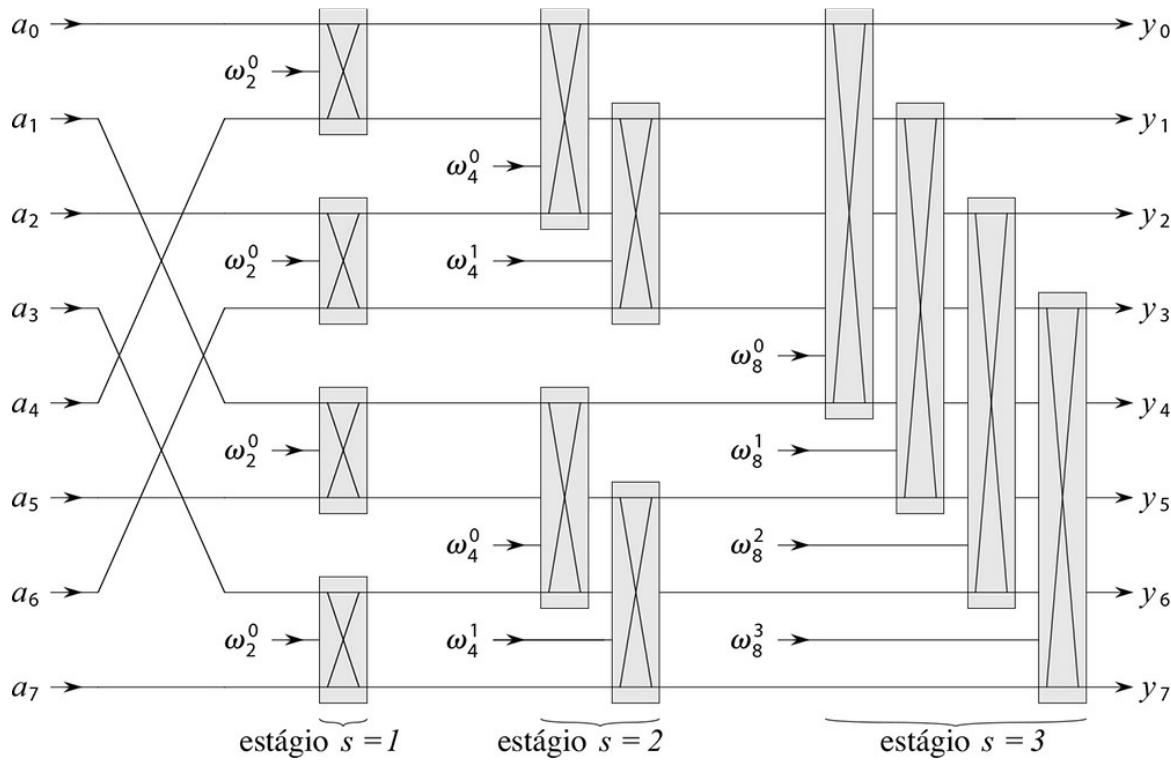
Podemos expressar o corpo do laço (linha 3) como pseudocódigo mais preciso. Copiamos o laço **for** do procedimento Recursive-Fft, identificando $y_{[0]}$ com $A[k \,..\, k + 2^{s-1} - 1]$ e $y_{[1]}$ com $A[k\ 2^{s-1} \,..\, k + 2^{s} - 1]$. O valor de giro usado em cada operação borboleta depende do valor de s; ele é uma potência de ^{m}, onde $m = 2$. (Introduzimos a variável m exclusivamente em atenção à legibilidade.) Introduzimos outra variável temporária u que nos permite executar a operação borboleta no lugar. Quando substituímos a linha 3 da estrutura global pelo corpo do laço, obtemos o pseudocódigo a seguir, que forma a base da implementação paralela que apresentaremos mais adiante. Primeiro o código chama o procedimento auxiliar Bit-Reverse-Copy(a, A) para copiar o vetor a para o arranjo A na ordem inicial em que precisamos dos valores.

```
ITERATIVE-FFT(a)
1  BIT-REVERSE-COPY(a, A)
2  n = a.comprimento          // n é uma potência de 2.
3  for s = 1 to lg n
4        m = 2^s
5        ω = e^{2πi/m}
6        for k = 0 to n − 1 by m
7              ω = 1
8              for j = 0 to m/2 − 1
9              t = ωA[k + j + m/2]
10             u = A[k + j]
11             A[k + j] = u + t
12             A[k + j + m/2] = u − t
13             ω = ωω_m
14  return A
```

Como Bit-Reverse-Copy obtém os elementos do vetor de entrada *a* na ordem desejada no arranjo *A*? A ordem em que as folhas aparecem na Figura 30.4 é uma ***permutação com inversão de bits***. Isto é, se representarmos por rev(*k*) o inteiro de lg *n* bits formado pela inversão dos bits da representação binária de *k*, teremos de inserir o elemento de vetor a_k na posição de arranjo *A*[rev(*k*)]. Na Figura 30.4, por exemplo, as folhas aparecem na ordem 0, 4, 2, 6, 1, 5, 3, 7; em linguagem binária, essa sequência é 000, 100, 010, 110, 001, 101, 011, 111 e, quando invertemos os bits de cada valor, obtemos a sequência 000, 001, 010, 011, 100, 101, 110, 111. Para ver que precisamos de uma permutação com inversão de bits em geral, observamos que, no nível superior da árvore, índices cujo bit de ordem baixa é 0 entram na subárvore à esquerda e índices cujo bit de ordem baixa é 1 entram na subárvore à direita. Extraindo o bit de ordem baixa em cada nível, continuamos esse processo descendo a árvore até obtermos a ordem dada pela permutação por reversão de bits nas folhas.

Visto que é fácil calcular a função rev(*k*), o procedimento Bit-Reverse-Copy é simples:

```
BIT-REVERSE-COPY(a, A)
1  n = a.comprimento
2  for k = 0 to n − 1
3        A[rev(k)] = a_k
```

A implementação de FFT iterativa é executada no tempo *Q*(*n* lg *n*). A chamada a Bit-Reverse-Copy(*a*, *A*) certamente é executada no tempo *O*(*n* lg *n*), já que iteramos *n* vezes e podemos inverter a representação de um inteiro entre 0 e *n* - 1, com lg *n* bits, no tempo *O*(lg *n*). (Na prática, como em geral, conhecemos o valor inicial de *n* com antecedência, provavelmente codificaríamos um mapeamento de tabela *k* para rev(*k*), fazendo Bit-Reverse-Copy ser executado no tempo *Q*(*n*) com uma constante oculta baixa. Como alternativa, poderíamos usar o esquema inteligente do contador binário com inversão de bits descrito no Problema 17-1.) Para concluir a prova de que Iterative-Fft é executado no tempo Q(*n* lg *n*), mostramos que *L*(*n*), o número de vezes que o corpo do laço mais interno (linhas 8-13) é executado, é Q(*n* lg *n*). O laço **for** das linhas 6-13 itera $n/m = n/2^s$ vezes para cada valor de *s*, e o laço mais interno das linhas 8-13 itera $m/2 = 2^{s-1}$ vezes. Assim,

$$L(n) = \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1}$$
$$= \sum_{s=1}^{\lg n} \frac{n}{2}$$
$$= \Theta(n \lg n)$$

Um circuito FFT paralelo

Podemos explorar muitas das propriedades que nos permitiram implementar um algoritmo FFT iterativo eficiente para produzir um algoritmo paralelo eficiente para a FFT. Expressaremos o algoritmo FFT paralelo como um circuito. A Figura 30.5 mostra um circuito FFT paralelo que calcula a FFT para n entradas, para n = 8. O circuito começa com uma permutação com inversão de bits das entradas, seguida por lg n estágios, cada estágio consistindo em $n/2$ borboletas executadas em paralelo. A ***profundidade*** do circuito — o número máximo de elementos computacionais entre qualquer saída e qualquer entrada que pode alcançá-lo — é, portanto, Q(lg n).

A parte da extrema esquerda do circuito FFT paralelo executa a permutação com inversão de bits, e o restante imita o procedimento iterativo Iterative-Fft. Como cada iteração do laço **for** mais externo executa $n/2$ operações borboleta independentes, o circuito as executa em paralelo. O valor de s em cada iteração dentro de Iterative-Fft corresponde a um estágio de borboletas, como mostra a Figura 30.5. Para s = 1, 2, ..., lg n, o estágio s consiste em $n/2^s$ grupos de borboletas (correspondentes a cada valor de k em Iterative-Fft), com 2^{s-1} borboletas por grupo (correspondentes a cada valor de j em Iterative-Fft). As borboletas mostradas na Figura 30.5 correspondem às operações borboleta do laço mais interno (linhas 9-12 de Iterative-Fft). Observe também que os fatores de giro usados nas borboletas correspondem aos que são utilizados em Iterative-Fft: no estágio s, usamos $\omega_m^0, \omega_m^1 \cdots \omega_m^{m/2-1}$, onde m = 2^s.

Figura 30.5 Um circuito que calcula a FFT em paralelo, mostrado aqui para $n = 8$ entradas. Cada operação borboleta toma como entrada os valores em dois fios, juntamente com um fator de giro, e produz como saídas os valores em dois fios. Os estágios de borboletas são identificados de modo a corresponder a iterações do laço mais externo do procedimento ITERATIVE-FFT. Somente os fios superior e inferior que passam por uma borboleta interagem com ela; fios que passam pelo meio de uma borboleta não afetam essa borboleta nem seus valores são alterados por essa borboleta. Por exemplo, a borboleta superior no estágio 2 não tem nenhuma relação com o fio 1 (o fio cuja saída é identificada por y_1); suas entradas e saídas estão apenas nos fios 0 e 2 (identificadas por y_0 e y_2, respectivamente). Esse circuito tem profundidade Q(lg n) e executa Q(n lg n) operações borboleta no total.

Exercícios

30.3-1 Mostre como Iterative FFT calcula a DFT do vetor de entrada (0, 2, 3, -1, 4, 5, 7, 9).

30.3-2 Mostre como implementar um algoritmo FFT com a permutação com inversão de bits ocorrendo no final e não no início do cálculo. (*Sugestão*: Considere a DFT inversa.)

30.3-3 Quantas vezes Iterative FFT calcula fatores de giro em cada fase? Reescreva Iterative FFT para calcular fatores de giro somente 2^{s-1} vezes na fase s.

30.3-4 ★ Suponha que os somadores dentro das operações borboleta no circuito FFT, às vezes, falham de tal maneira que sempre produzem uma saída zero, independentemente de suas entradas. Suponha que exatamente um somador tenha falhado, mas que você não saiba qual. Descreva como é possível identificar o somador que falhou fornecendo entradas para o circuito FFT global e observando as saídas. Qual é a eficiência do seu método?

Problemas

30-1 ***Multiplicação por divisão e conquista***

a. Mostre como multiplicar dois polinômios lineares $ax + b$ e $cx + d$ usando apenas três multiplicações. (*Sugestão*: Uma das multiplicações é $(a + b) \cdot (c + d)$.)

b. Dê dois algoritmos de divisão e conquista para multiplicar dois polinômios de grau limitado por n no tempo $Q(n^{\lg 3})$. O primeiro algoritmo deve dividir os coeficientes dos polinômios de entrada em uma metade alta e uma metade baixa, e o segundo algoritmo deve dividi-los conforme o índice seja ímpar ou par.

c. Mostre como multiplicar dois inteiros de n bits em $Q(n^{\lg 3})$ etapas, em que cada etapa opera no máximo em um número constante de valores de 1 bit.

30-2 Matrizes de Toeplitz

Uma ***matriz de Toeplitz*** é uma matriz $n \times n$ $A = (a_{ij})$ tal que $a_{ij} = a_{i-1, j-1}$ para $i = 2, 3, ..., n$ e $j = 2, 3, ..., n$.

a. A soma de duas matrizes de Toeplitz é necessariamente de Toeplitz? E o produto?

b. Descreva como representar uma matriz de Toeplitz de modo que você possa somar duas matrizes de Toeplitz $n \times n$ no tempo $O(n)$.

c. Dê um algoritmo de tempo $O(n \lg n)$ para multiplicar uma matriz de Toeplitz $n \times n$ por um vetor de comprimento n. Use sua representação da parte (b).

d. Dê um algoritmo eficiente para multiplicar duas matrizes de Toeplitz $n \times n$. Analise o tempo de execução.

30-3 Transformada rápida de Fourier multidimensional

Podemos generalizar a transformada discreta de Fourier unidimensional definida pela equação (30.8) para d dimensões. A entrada é um arranjo d-dimensional $A = (a_{j_1, j_2, \ldots, j_d})$ cujas dimensões são $n_1, n_2, ..., n_d$, onde $n_1 n_2 \ldots n_d = n$. Definimos a transformada discreta de Fourier d-dimensional pela equação

$$y_{k_1, k_2, \ldots, k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1, j_2, \ldots, j_d} \, \omega_{n_2}^{j_1 k_1} \cdots \omega_{n_d}^{j_d k_d}$$

$$\text{para } 0 \le k_1 < n_1, 0 \le k_2 < n_2, ..., 0 \le k_d < n_d.$$

a. Mostre que podemos calcular uma DFT d-dimensional calculando DFTs unidimensionais em cada dimensão por vez. Isto é, primeiro calculamos n/n_1 DFTs unidimensionais separadas ao longo da dimensão 1. Então, usando o resultado das DFTs ao longo da dimensão 1 como entrada, calculamos n/n_2 DFTs unidimensionais separadas ao longo da dimensão 2. Usando esse resultado como entrada, calcule n/n_3 DFTs unidimensionais separadas ao longo da dimensão 3, e assim por diante, até a dimensão d.

b. Mostre que a ordenação de dimensões não importa, de modo que podemos calcular uma DFT d-dimensional calculando as DFTs unidimensionais em qualquer ordem das d dimensões.

c. Mostre que, se calcularmos cada DFT unidimensional calculando a transformada rápida de Fourier, o tempo total para calcular uma DFT d-dimensional é $O(n \lg n)$, independentemente de d.

30-4 Avaliar todas as derivadas de um polinômio em um ponto

Dado um polinômio $A(x)$ de grau limitado a n, definimos sua t-ésima derivada por

$$A^{(t)}(x) = \begin{cases} A(x) & \text{se } t = 0\,, \\ \frac{d}{dx} A^{(t-1)} & \text{se } 1 \le t \le n-1\,, \\ 0 & \text{se } t \ge n\,. \end{cases}$$

Pela representação por coeficientes $(a_0, a_1, ..., a_{n-1})$ de $A(x)$ e um ponto dado x_0, desejamos determinar $A_{(t)}(x)$ para $t = 0, 1, ..., n - 1$.

a. Dados os coeficientes $b_0, b_1, ..., b_{n-1}$ tais que

$$A^{(t)}(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j\,,$$

mostre como calcular $A_{(t)}(x_0)$ para $t = 0, 1, ..., n - 1$, no tempo $O(n)$.

b. Explique como determinar $b_0, b_1, ..., b_{n-1}$ no tempo $O(n \lg n)$, dado $A(x + \omega_k)$ para $k = 0,1,...,n - 1$.

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right),$$

onde $f(j) = a_j \cdot j!$ e

$$g(l) = \begin{cases} x_0^{-l} / (-l)! & se -(n-1) \le l \le 0, \\ 0 & se\ 1 \le l \le n-1. \end{cases}$$

c. Explique como avaliar $A(x_0 +_n)$ para $k = 0, 1, ..., n - 1$ no tempo $O(n \lg n)$. Conclua que podemos avaliar todas as derivadas não triviais de $A(x)$ em x_0 no tempo $O(n \lg n)$.

30-5 Avaliar polinômios em vários pontos

Já vimos como avaliar um polinômio de grau limitado por $n - 1$ em um ponto isolado no tempo $O(n)$ usando a regra de Horner. Também descobrimos como avaliar tal polinômio em todas as n raízes complexas da unidade no tempo $O(n \lg n)$ usando a FFT. Agora, mostraremos como avaliar um polinômio de grau limitado por n em n pontos arbitrários no tempo $O(n \lg^2 n)$.

Para tal, suporemos que podemos calcular o resto do polinômio quando um desses polinômios é dividido por outro no tempo $O(n \lg n)$, um resultado que afirmamos sem prova. Por exemplo, o resto de $3x_3 + x_2 - 3x + 1$ quando dividido por $x_2 + x + 2$ é

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = -7x + 5\,.$$

Dada a representação por coeficientes de um polinômio $A(x) = \sum_{k=0}^{n-1} a_k x^k$ e n pontos $x_0, x_1, ..., x_{n-1}$, desejamos calcular os n valores $A(x_0), A(x_1), ..., A(x_{n-1})$. Para $0 \leq i \leq j \leq n-1$, defina os polinômios $P_{ij}(x) = \prod_{k=1}^{j} (x - x_k)$ e $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$. Observe que $Q_{ij}(x)$ tem grau no máximo $j - i$.

a. Prove que $A(x) \bmod (x - z) = A(z)$ para qualquer ponto z.

b. Prove que $Q_{kk}(x) = A(x_k)$ e que $Q_{0,n-1}(x) = A(x)$.

c. Prove que, para $i \leq k \leq j$, temos $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ e $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

d. Dê um algoritmo de tempo $O(n \lg^2 n)$ para avaliar $A(x\ ), A(x\ ), ..., A(x\ )$.

30-6 **FFT *com aritmética modular***

Conforme definido, a transformada discreta de Fourier requer cálculo com números complexos, o que pode resultar em uma perda de precisão devido a erros de arredondamento. Para alguns problemas, sabe-se que a resposta contém somente inteiros e, se usarmos uma variante da FFT baseada em aritmética modular, poderemos garantir que a resposta será calculada com exatidão. Um exemplo de tal problema é o de multiplicar dois polinômios com coeficientes inteiros. O Exercício 30.2-6 nos dá uma abordagem, usando um módulo de comprimento $\Omega(n)$ bits para tratar uma DFT em n pontos. Este problema apresenta uma outra abordagem que usa um módulo com o comprimento mais razoável $O(\lg n)$. O problema requer que você entenda o material do Capítulo 31. Seja n uma potência de 2.

a. Suponha que procuramos o menor k tal que $p = kn + 1$ é primo. Dê um argumento heurístico simples que explique por que poderíamos esperar que k seja aproximadamente $\ln n$. (O valor de k poderia ser muito maior ou muito menor, mas é razoável esperar que examinaremos $O(\lg n)$ valores candidatos de k em média.) Como o comprimento esperado de p se compara com o de n?

Seja g um gerador de $_p$* e seja $w = g_k \bmod p$.

b. Demonstre que a DFT e a DFT inversa são operações inversas bem definidas módulo p, onde w é usado como uma raiz n-ésima principal da unidade.

c. Demonstre como fazer com que a FFT e sua inversa funcionem em módulo p no tempo $O(n \lg n)$, onde operações em palavras de $O(\lg n)$ bits demoram um tempo unitário. Considere que o algoritmo conhece p e w.

d. Calcule a DFT módulo $p = 17$ do vetor (0, 5, 3, 7, 7, 2, 1, 6). Observe que $g = 3$ é um gerador de $*_{17}$.

NOTAS DO CAPÍTULO

O livro de Van Loan [343] dá um tratamento extraordinário da transformada rápida de Fourier. Press, Teukolsky, Vetterling e Flannery [283, 284] apresentam uma boa descrição da transformada rápida de Fourier e suas aplicações. Se quiser uma excelente introdução ao processamento de sinais, uma área bem conhecida de aplicação de FFT, consulte os textos de Oppenheim e Schafer [266] e de Oppenheim e Willsky [267]. O livro de Oppenheim e Schafer também mostra como tratar casos em que n não é uma potência inteira de 2.

A análise de Fourier não é limitada a dados unidimensionais. Ela é amplamente usada em processamento de imagens para analisar dados em duas ou mais dimensões. Os livros de Gonzalez e Woods [146] e Pratt [281] discutem transformadas de Fourier multidimensionais e seu uso em processamento de imagens, e os livros de Tolimieri, An e Lu [338] e Van Loan [343] discutem os fundamentos matemáticos de transformadas rápidas de Fourier.

Cooley e Tukey [77] são amplamente reconhecidos como os criadores da FFT na década de 1960. Na verdade, a FFT foi descoberta muito antes dessa data, mas sua importância só foi completamente percebida após o advento dos computadores digitais modernos. Embora Press, Teukolsky, Vetterling e Flannery atribuam as origens do método a Runge e König em 1924, um artigo de Heideman, Johnson e Burrus [163] afirma que o histórico da FFT remonta à época de C. F. Gauss, em 1805.

Frigo e Johnson [117] desenvolveram uma implementação rápida e flexível da FFT, denominada FFTW ("a transformada de Fourier mais rápida do oeste"). A FFTW é projetada para situações que exigem vários cálculos de DFT para o mesmo tamanho de problema. Antes de calcular as DFTs, a FFTW executa um "planejador" que, por uma série de execuções experimentais, determina como decompor melhor o cálculo da FFT para o tamanho do problema dado no computador hospedeiro. A FFTW adapta-se para utilizar a cache do hardware eficientemente e, desde que os problemas sejam suficientemente pequenos, a FFTW os resolve com um código otimizado, sem laços. Além disso, a FFTW tem a vantagem incomum de demorar o tempo Q (n lg n) para qualquer tamanho n de problema, mesmo quando n é um número primo grande. Embora a transformada de Fourier padrão supõe que a entrada representa pontos uniformemente espaçados no domínio do tempo, outras técnicas podem aproximar a FFT para dados que não são uniformemente espaçados (equiespaçados). O artigo de Ware [348] dá uma visão geral.

---

[1] A interpolação é um problema notoriamente complicado do ponto de vista da estabilidade numérica. Embora as abordagens descritas aqui sejam matematicamente corretas, pequenas diferenças nas entradas ou erros de arredondamento durante o cálculo podem gerar grandes diferenças no resultado.

[2] Muitos outros autores dão uma definição diferente de $_n$: $e^{-2pi/n}$. Essa definição alternativa tende a ser usada para aplicações de processamento de sinais. A matemática subjacente é substancialmente a mesma com qualquer das definições de $_n$.

[3] O comprimento n é, na realidade, o que denominamos $2n$ na Seção 30.1, visto que dobramos a limitação de grau dos polinômios dados antes da avaliação. Portanto, no contexto da multiplicação de polinômios, na verdade estamos trabalhando com raízes $(2n)$-ésimas complexas da unidade.

31 ALGORITMOS DA TEORIA DOS NÚMEROS

A teoria dos números já foi vista como um assunto belo, mas em grande parte inútil na matemática pura. Hoje, os algoritmos da teoria dos números são amplamente utilizados devido, em parte, à criação de esquemas criptográficos baseados em números primos grandes. Esses esquemas são viáveis porque é fácil encontrar primos grandes, e são seguros porque não sabemos como fatorar eficientemente o produto de primos grandes ou resolver outros problemas relacionados, como calcular logaritmos discretos. Este capítulo apresenta alguns dos algoritmos da teoria dos números e associados subjacentes a tais aplicações.

A Seção 31.1 apresenta conceitos básicos da teoria dos números, como divisibilidade, equivalência modular e fatoração única. A Seção 31.2 estuda um dos algoritmos mais antigos do mundo: o algoritmo de Euclides para calcular o máximo divisor comum de dois inteiros. A Seção 31.3 revê conceitos de aritmética modular. A Seção 31.4 estuda o conjunto de múltiplos de um dado número a, módulo n e mostra como determinar todas as soluções para a equação $ax \equiv b \pmod{n}$ usando o algoritmo de Euclides. O teorema chinês do resto é apresentado na Seção 31.5. A Seção 31.6 considera potências de um dado número a, módulo n e apresenta um algoritmo de elevação ao quadrado repetido para calcular eficientemente $a_b \bmod n$, dados a, b e n. Essa operação está no núcleo do teste de primalidade eficiente e em grande parte da criptografia moderna. Em seguida, a Seção 31.7 descreve o sistema de criptografia de chave pública RSA. A Seção 31.8 examina um teste de primalidade aleatorizado. Podemos usar esse teste para determinar primos eficientemente, tarefa necessária para criar chaves para o sistema de criptografia RSA. Finalmente, a Seção 31.9 revê uma heurística simples, mas efetiva para fatorar inteiros pequenos. Não deixa de ser curioso que a fatoração é um problema que as pessoas gostariam que fosse intratável, já que a segurança do RSA depende da dificuldade de fatorar inteiros grandes.

Tamanho de entradas e custo de cálculos aritméticos

Como trabalharemos com inteiros grandes, precisamos ajustar nosso modo de pensar sobre o tamanho de uma entrada e sobre o custo de operações aritméticas elementares.

Neste capítulo, "entrada grande" em geral significa uma entrada que contenha "inteiros grandes", em vez de uma entrada que contenha "muitos inteiros" (como na ordenação). Assim, mediremos o tamanho de uma entrada em termos do *número de bits* exigidos para representar essa entrada, não apenas em termos do número de inteiros na entrada. Um algoritmo com entradas inteiras $a_1, a_2, ..., a_k$ é um ***algoritmo de tempo polinomial*** se for executado em tempo polinomial em relação a $\lg a_1, \lg a_2, ..., \lg a_k$, isto é, polinomial em relação aos comprimentos de suas entradas codificadas em binário.

Na maior parte deste livro, verificamos que é conveniente pensar em operações aritméticas elementares (multiplicações, divisões ou cálculo de restos) como operações primitivas que demoram uma unidade de tempo. Contando o número dessas operações aritméticas que um algoritmo efetua, temos uma base para fazer uma estimativa razoável do tempo de execução real do algoritmo em um computador. Contudo, operações elementares podem ser demoradas quando suas entradas são grandes. Assim, torna-se conveniente medir quantas ***operações com bits*** um algoritmo da teoria dos números exige. Nesse modelo, multiplicar dois inteiros de bits pelo método comum utiliza (2)

operações com bits. De modo semelhante, podemos dividir um inteiro de bits por um inteiro mais curto ou tomar o resto da divisão de um inteiro de bits quando dividido por um inteiro mais curto no tempo (^{2}) por algoritmos simples (veja o Exercício 31.1-12). São conhecidos métodos mais rápidos. Por exemplo, um método simples de divisão e conquista para multiplicar dois inteiros de bits tem um tempo de execução ($^{\lg 3}$) e o método mais rápido conhecido tem um tempo de execução (lg lg lg). Porém, para finalidades práticas, o algoritmo (^{2}) frequentemente é melhor, e utilizaremos esse limite como base para nossas análises.

Neste capítulo, geralmente analisamos algoritmos em termos do número de operações aritméticas e também do número de operações com bits que eles exigem.

31.1 NOÇÕES DA TEORIA ELEMENTAR DOS NÚMEROS

Esta seção apresenta uma breve revisão de noções da teoria elementar dos números concernentes ao conjunto = {..., –2, –1, 0, 1, 2, ...} de inteiros e ao conjunto = {0, 1, 2, ...} de números naturais.

Divisibilidade e divisores

A noção de que um inteiro é divisível por outro é fundamental na teoria dos números. A notação $d \mid a$ (leia "*d* ***divide*** *a*") significa que $a = kd$ para algum inteiro k. Todo inteiro divide 0. Se $a > 0$ e $d \mid a$, então $|d| \leq |a|$. Se $d \mid a$, então dizemos também que a é um ***múltiplo*** de d. Se d não divide a, escrevemos $d \nmid a$.

Se $d \mid a$ e $d \geq 0$, dizemos que d é um ***divisor*** de a. Note que $d \mid a$ se e somente se $-d \mid a$, de modo que não há nenhuma perda de generalidade se definirmos os divisores como não negativos, entendendo-se que o negativo de qualquer divisor de a também divide a. Um divisor de um inteiro não nulo a é no mínimo 1, mas não maior que $|a|$. Por exemplo, os divisores de 24 são 1, 2, 3, 4, 6, 8, 12 e 24.

Todo inteiro positivo a é divisível pelos ***divisores triviais*** 1 e a. Os divisores não triviais de a são os ***fatores*** de a. Por exemplo, os fatores de 20 são 2, 4, 5 e 10.

Números primos e compostos

Um inteiro $a > 1$ cujos únicos divisores são os divisores triviais 1 e a é chamado ***número pri- mo*** (ou, mais simplesmente, ***primo***). Primos têm muitas propriedades especiais e desempenham um papel fundamental na teoria dos números. Os primeiros 20 primos, em ordem crescente, são

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71.$$

O Exercício 31.1-2 pede que você prove que existe um número infinito de primos. Um inteiro $a > 1$ que não é primo é denominado ***número composto*** (ou, mais simplesmente, ***composto***). Por exemplo, 39 é composto porque $3 \mid 39$. O inteiro 1 é denominado ***unidade*** e não é primo nem composto. De modo semelhante, o inteiro 0 e todos os inteiros negativos não são primos nem compostos.

Teorema da divisão, restos e equivalência modular

Dado um inteiro n, podemos repartir os inteiros entre os que são múltiplos de n e os que não são múltiplos de n. Grande parte da teoria dos números é baseada em um refinamento dessa partição pela classificação dos não múltiplos de n de acordo com seus restos quando divididos por n. O teorema a seguir é a base para esse refinamento. Omitimos a prova (mas veja, por exemplo, Niven e Zuckerman [265]).

Teorema 31.1 (Teorema da divisão)

Para qualquer inteiro a e qualquer inteiro positivo n, existem inteiros únicos q e r tais que $0 \leq r <$ e $a = qn + r$.

O valor $q =$ a/n é o ***quociente*** da divisão. O valor $r = a$ mod n é o ***resto*** da divisão. Temos que $n \mid a$ se e somente se a mod $n = 0$.

Podemos repartir os inteiros em n classes de equivalência, de acordo com seus restos módulo n. A ***classe de equivalência módulo n*** que contém um inteiro a é

$$[a]_n = \{a + kn : k \in \mathbb{Z}\} .$$

Por exemplo, $[3]_7 = \{..., -11, -4, 3, 10, 17, ...\}$; também podemos representar esse conjunto por $[-4]_7$ e $[10]_7$. Usando a notação definida na página 40, podemos dizer que escrever $a \in [b]_n$ é o mesmo que escrever $a \equiv b$. O conjunto de todas essas classes de equivalência é

$$\mathbb{Z}_n = \{[a]_n : 0 \leq a \leq n - 1\} \quad (31.1)$$

Quando vir a definição

$$\mathbb{Z}_n = \{0, 1, ..., \mathrm{n} - 1\} , \quad (31.2)$$

você a deve ler como equivalente à equação (31.1), entendendo que 0 representa $[0]_n$, 1 representa $[1]_n$, e assim por diante; cada classe é representada por seu menor elemento não negativo. Contudo, você deve ter em mente as classes de equivalência subjacentes. Por exemplo, se nos referirmos a – 1 como um membro de $_n$, na verdade estamos nos referindo a $[n - 1]_n$, já que $-1 \equiv n - 1 \pmod{n}$.

Divisores comuns e máximos divisores comuns

Se d é um divisor de a e também um divisor de b, então d é um ***divisor comum*** de a e b. Por exemplo, os divisores de 30 são 1, 2, 3, 5, 6, 10, 15 e 30, e portanto os divisores comuns de 24 e 30 são 1, 2, 3 e 6. Observe que 1 é um divisor comum de quaisquer dois inteiros.

Uma propriedade importante dos divisores comuns é que

$$d \mid a \text{ e } d \mid b \text{ implica } d \mid (a + b) \text{ e } d \mid (a - b) \quad (31.3)$$

De modo mais geral, temos que

$$d \mid a \text{ e } d \mid b \text{ implica } d \mid (ax + by) \quad (31.4)$$

para quaisquer inteiros x e y. Além disso, se $a \mid b$, então $|a| \leq |b|$ ou $b = 0$, o que implica que

$$a \mid b \text{ e } b \mid a \text{ implica } a = \pm\, b . \quad (31.5)$$

O ***máximo divisor comum*** de dois inteiros a e b, ambos não nulos, é o maior dos divisores comuns de a e b e é denotado por mdc(a, b). Por exemplo, mdc(24, 30) = 6, mdc(5, 7) = 1 e mdc(0, 9) = 9. Se a e b são não nulos, então mdc(a, b) é um inteiro entre 1 e min($|a|$, $|b|$). Definimos mdc(0, 0) como 0; essa definição é necessária para que as propriedades padrões da função mdc (como a equação (31.9) a seguir) sejam universalmente válidas.

As seguintes são propriedades elementares da função mdc:

$$\begin{aligned}
\mathrm{mdc}(a, b) &= \mathrm{mdc}(b, a) , && (31.6)\\
\mathrm{mdc}(a, b) &= \mathrm{mdc}\,(-a, b) , && (31.7)\\
\mathrm{mdc}(a, b) &= \mathrm{mdc}(|a|, |b|) , && (31.8)\\
\mathrm{mdc}(a, 0) &= |a| , && (31.9)\\
\mathrm{mdc}(a, ka) &= |a| \text{ para qualquer } k \in \mathbb{Z}. && (31.10)
\end{aligned}$$

O teorema apresentado a seguir dá uma caracterização alternativa e útil de mdc(a, b).

Teorema 31.2

Se a e b são inteiros quaisquer e não nulos, então mdc(a, b) é o menor elemento positivo do conjunto $\{ax + by : x, y \in \}$ de combinações lineares de a e b.

Prova Seja s a menor combinação positiva dessas combinações lineares de a e b, e seja $s = ax + by$ para algum $x, y \in$. Seja $q =\ \ a/s$. Então a equação (3.8) implica

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a(1 - qx) + b(-qy) , \end{aligned}$$

e, portanto, $a \bmod s$ também é uma combinação linear de a e b. Porém, como $0 \leq a$, temos que $a \bmod s = 0$ porque s é a menor combinação positiva dessas combinações lineares. Portanto, temos que $s \mid a$ e, por raciocínio análogo, $s \mid b$. Assim, s é um divisor comum de a e b, então mdc(a, b) $\geq s$. A equação (31.4) implica que mdc(a, b) $\mid s$, visto que mdc(a, b) divide a e b, e s é uma combinação linear de a e b. Porém, mdc(a, b) $\mid s$ e $s > 0$ implicam que mdc(a, b) $\leq s$. Combinando mdc(a, b) $\geq s$ e mdc(a, b) $\leq s$ obtemos mdc(a, b) $= s$. Concluímos que s é o máximo divisor comum de a e b.

Corolário 31.3

Para quaisquer inteiros a e b, se $d \mid a$ e $d \mid b$, então $d \mid$ mdc(a, b).

Prova Esse corolário decorre da equação (31.4) porque mdc(a, b) é uma combinação linear de a e b pelo Teorema 31.2.

Corolário 31.4

Para todos os inteiros a e b e qualquer inteiro não negativo n,

$$\text{mdc}(an, bn) = n \text{ mdc}(a, b).$$

Prova Se $n = 0$, o corolário é trivial. Se $n > 0$, então mdc(an, bn) é o menor elemento positivo do conjunto $\{anx + bny : x, y, \text{ e } \}$, que é n vezes o menor elemento positivo do conjunto $\{ax + by: x, y, \text{ e } \}$.

Corolário 31.5

Para todos os inteiros positivos n, a e b, se $n \mid ab$ e mdc(a, n) = 1, então $n \mid b$.

Prova Deixamos a prova para o Exercício 31.1-5.

Inteiros primos entre si

Dois inteiros a e b são ***primos entre si*** (também: cada um é ***primo com*** o outro) se seu único divisor comum é 1, isto é, se mdc(a, b) = 1. Por exemplo, 8 e 15 são primos entre si já que os divisores de 8 são 1, 2, 4 e 8, enquanto os divisores de 15 são 1, 3, 5 e 15. O teorema a seguir afirma que, se cada um de dois inteiros e um inteiro p são primos entre si, então seu produto e p são primos entre si.

Teorema 31.6

Para quaisquer inteiros a, b e p, se mdc(a, p) = 1 e mdc(b, p) = 1, então mdc(ab, p) = 1.

Prova Decorre do Teorema 31.2 que existem inteiros x, y, x' e y' tais que

$$ax + py = 1 ,$$
$$bx' + py' = 1 .$$

Multiplicando essas equações e reorganizando, temos

$$ab(xx') + p(ybx' + y'ax + pyy') = 1 .$$

Assim, visto que 1 é uma combinação linear positiva de ab e p, usamos o Teorema 31.2 para concluir a prova.

Os inteiros n_1, n_2, ..., n_k são ***primos dois a dois*** se, sempre que $i \neq j$, temos mdc(n_i, n_j) = 1.

Fatoração única

Um fato elementar mas importante sobre a divisibilidade por primos é dado a seguir.

Teorema 31.7

Para todos os primos p e todos os inteiros a, b, se $p \mid ab$, então $p \mid a$ ou $p \mid b$ (ou ambos).

Prova Suponha, por contradição, que $p \mid ab$, mas que p a e p b. Assim, mdc(a, p) = 1 e mdc(b, p) = 1, já que os únicos divisores de p são 1 e p, e estamos supondo que p não divide a nem b. Então, o Teorema 31.6 implica que mdc(ab, p) = 1, contradizendo nossa hipótese de que $p \mid ab$, já que $p \mid ab$ implica mdc(ab, p) = p. Essa contradição conclui a prova.

Uma consequência do Teorema 31.7 é que um inteiro tem uma fatoração única em primos.

Teorema 31.8 (Fatoração única)

Há exatamente um modo de escrever qualquer inteiro composto a como um produto da forma

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} ,$$

onde os p_i são primos, $p_1 < p_2 < \ldots < p_r$, e os e_i são inteiros positivos.

Prova Deixamos a prova para o Exercício 31.1-11.

Como exemplo, o número 6.000 pode ser fatorado unicamente em primos como $2^4 \cdot 3 \cdot 5^3$.

Exercícios

31.1-1 Prove que, se $a > b > 0$ e $c = a + b$, então $c \bmod a = b$.

31.1-2 Prove que existe um número infinito de primos. (*Sugestão*: Mostre que nenhum dos primos p_1, p_2, ..., p_k divide $(p_1p_2 \ldots p_k) + 1$.)

31.1-3 Prove que, se $a \mid b$ e $b \mid c$, então $a \mid c$.

31.1-4 Prove que, se p é primo e $0 < k < p$, então mdc(k, p) = 1.

31.1-5 Prove o Corolário 31.5.

31.1-6

Prove que, se p é primo e $0 < k < p$, então P/ $\binom{P}{k}$. Conclua que para todos os inteiros a, b e primos p, $(a + b)^p \equiv a_p + b_p$ (mod p) .

31.1-7 Prove que, se a e b são quaisquer inteiros positivos tais que $a \mid b$, então (x mod b) mod a = x mod a para qualquer x. Prove, sob as mesmas hipóteses, que $x \equiv y$ (mod b) implica $x \equiv y$ (mod a) para quaisquer inteiros x e y.

31.1-8 Para qualquer inteiro $k > 0$, dizemos que um inteiro n é uma ***k-ésima potência*** se existe um inteiro a tal que $a_k = n$. Além disso, $n > 1$ é uma ***potência não trivial*** se é uma k-ésima potência para algum inteiro $k > 1$. Mostre como determinar se um determinado inteiro n de bits é uma potência não trivial em tempo polinomial em .

31.1-9 Prove as equações (31.6)–(31.10).

31.1-10 Mostre que o operador mdc é associativo. Isto é, prove que, para todos os inteiros a, b e c, mdc(a, mdc(b, c)) = mdc(mdc(a, b), c)

31.1-11 ★ Prove o Teorema 31.8.

31.1-12 Dê algoritmos eficientes para as operações de dividir um inteiro de bits por um inteiro mais curto e de tomar o resto da divisão de um inteiro de bits por um inteiro mais curto. Seus algoritmos devem ser executados no tempo (^{2}).

31.1-13 Dê um algoritmo eficiente para converter um determinado inteiro (binário) de bits para representação decimal. Demonstre que, se a multiplicação ou divisão de inteiros cujo comprimento é no máximo demorar o tempo M(), então podemos converter de binário para decimal no tempo (M() lg). (*Sugestão*: Use uma abordagem de divisão e conquista, obtendo as metades superior e inferior do resultado com recursões separadas.)

31.2 MÁXIMO DIVISOR COMUM

Nesta seção, descrevemos o algoritmo de Euclides para calcular eficientemente o máximo divisor comum de dois inteiros. Quando analisarmos o tempo de execução, veremos uma conexão surpreendente com os números de Fibonacci, que produz uma entrada do pior caso para o algoritmo de Euclides.

Nesta seção, nos limitaremos a inteiros não negativos. Essa restrição é justificada pela equação (31.8), que declara que mdc(a, b) = mdc($|a|$, $|b|$).

Em princípio, podemos calcular mdc(a, b) para inteiros positivos a e b pelos fatores primos de a e b. De fato, se

$$a = P_1^{e_1} P_2^{e_2} \cdots P_r^{e_r} \tag{31.11}$$

$$a = P_1^{f_1} P_2^{f_2} \cdots P_r^{f_r} \tag{31.12}$$

sendo que expoentes nulos são usados para tornar o conjunto de primos $p_1, p_2, ..., p_r$ igual para a e b, então, como o Exercício 31.2-1 pede para mostrar,

$$\text{mdc}(a,b) = p_1^{\min(e_1,f_1)} p_2^{\min(e_2,f_2)} \ldots P_r^{(e_r,f_r)} \tag{31.13}$$

Porém, como mostraremos na Seção 31.9, os melhores algoritmos existentes para fatoração não são executados em tempo polinomial. Assim, parece improvável que essa abordagem para calcular o máximo divisor comum produza um algoritmo eficiente.

O algoritmo de Euclides para calcular máximos divisores comuns é baseado no teorema a seguir.

Teorema 31.9 (Teorema de recursão para o MDC)

Para qualquer inteiro não negativo a e qualquer inteiro positivo b,

$$\text{mdc}(a, b) = \text{mdc}(b, a \bmod b).$$

Prova Mostraremos que mdc(a, b) e mdc(b, a mod b) são divisíveis entre si, de modo que, pela equação (31.5), eles devem ser iguais (já que ambos são não negativos).

Primeiro, mostramos que mdc(a, b) | mdc(b, a mod b). Se fizermos d = mdc(a, b), então $d \mid a$ e $d \mid b$. Pela equação (3.8), a mod $b = a - qb$, onde $q = a/b$. Visto que a mod b é uma combinação linear de a e b, a equação (31.4) implica que $d \mid (a \bmod b)$. Portanto, visto que $d \mid b$ e $d \mid (a \bmod b)$, o Corolário 31.3 implica que d | mdc(b, a mod b) ou, o que é equivalente,

$$\text{mdc}(a, b) \mid \text{mdc}(b, a \bmod b) \tag{31.14}$$

Mostrar que mdc(b, a mod b) | mdc(a, b) é quase a mesma coisa. Se agora fizermos d = mdc(b, a mod b), então $d \mid b$ e $d \mid (a \bmod b)$. Visto que $a = qb + (a \bmod b)$, onde $q = a/b$, temos que a é uma combinação linear de b e (a mod b). Pela equação (31.4), concluímos que $d \mid a$. Como $d \mid b$ e $d \mid a$, temos que d | mdc(a, b) pelo Corolário 31.3 ou, o que é equivalente,

$$\text{mdc}(b, a \bmod b) \mid \text{mdc}(a, b). \tag{31.15}$$

Usando a equação (31.5) para combinar as equações(31.14) e (31.15) concluímos a prova.

Algoritmo de Euclides

A obra *Elementos*, de Euclides (aproximadamente 300 a.C.), descreve o algoritmo para mdc que apresentamos a seguir, embora sua origem talvez seja ainda mais remota. Expressamos o algoritmo de Euclides como um programa recursivo baseado diretamente no Teorema 31.9. As entradas a e b são inteiros não negativos arbitrários.

```
EUCLID(a, b)
1  if b == 0
2      return a
3  else return EUCLID(b, a mod b)
```

Como exemplo da execução de EUCLID, considere o cálculo de mdc(30, 21):

$$\begin{aligned}
\text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\
&= \text{EUCLID}(9, 3) \\
&= \text{EUCLID}(3, 0) \\
&= 3 .
\end{aligned}$$

Esse cálculo chama EUCLID recursivamente três vezes.

A correção de EUCLID decorre do Teorema 31.9 e da seguinte propriedade: se o algoritmo retorna a na linha 2, então $b = 0$, de modo que a equação (31.9) implica que mdc(a, b) = mdc(a, 0) = a. O algoritmo não pode executar recursões indefinidamente, já que o segundo argumento diminui estritamente em cada chamada recursiva e é sempre não negativo. Assim, EUCLID sempre termina com a resposta correta.

Tempo de execução do algoritmo de Euclides

Analisamos o tempo de execução do pior caso de EUCLID em função do tamanho de a e b. Supomos, sem perda de generalidade, que $a > b \geq 0$. Para justificar essa hipótese, observe que , se $b > a \geq 0$, então EUCLID(a, b) faz imediatamente a chamada recursiva EUCLID(b, a). Isto é, se o primeiro argumento é menor que o segundo argumento, EUCLID gasta uma chamada recursiva permutando seus argumentos e depois prossegue. De modo semelhante, se $b = a > 0$, o procedimento termina após uma chamada recursiva, já que a mod b = 0.

O tempo de execução global de EUCLID é proporcional ao número de chamadas recursivas que ele executa. Nossa análise usa os números de Fibonacci F_k, definidos pela recorrência (3.22).

Lema 31.10

Se $a > b \geq 1$ e a chamada EUCLID(a, b) executa $k \geq 1$ chamadas recursivas, então $a \geq F_{k+2}$ e $b \geq F_{k+1}$.

Prova A prova é por indução em relação a k. Para a base da indução, seja $k = 1$. Então, $b \geq 1 = F_2$ e, como $a > b$, devemos ter $a \geq 2 = F_3$. Visto que $b > (a \bmod b)$, em cada chamada recursiva o primeiro argumento é estritamente maior que o segundo; a hipótese $a > b$ é válida para cada chamada recursiva.

Suponha indutivamente que o lema vale se forem executadas $k - 1$ chamadas recursivas; então, provaremos que o lema vale para k chamadas recursivas. Visto que $k > 0$, temos $b > 0$, e EUCLID (a, b) chama EUCLID(b, a mod b) recursivamente que, por sua vez, efetua $k - 1$ chamadas recursivas. Então, a hipótese de indução implica que $b \geq F_{k+1}$ (provando assim uma parte do lema) e $a \bmod b \geq F_k$. Temos

$$\begin{aligned}
b + (a \bmod b) &= b + (a - b\lfloor a/b \rfloor) \\
&\leq a ,
\end{aligned}$$

visto que $a > b > 0$ implica $a/b \geq 1$. Assim,

$$\begin{aligned}
a &\geq b + (a \bmod b) \\
&\geq F_{k+1} + F_k \\
&= F_{k+2}
\end{aligned}$$

O teorema a seguir é um corolário imediato desse lema.

Teorema 31.11 (Teorema de Lamé)

Para qualquer inteiro $k \geq 1$, se $a > b \geq 1$ e $b < F_{k+1}$, então a chamada EUCLID(a, b) faz menos de k chamadas recursivas.

Podemos mostrar que o limite superior do Teorema 31.11 é o melhor possível mostrando que a chamada EUCLID(F_{k+1}, F_k) faz exatamente $k - 1$ chamadas recursivas quando $k \geq 2$. Usamos indução em relação a k. Para o caso-base, $k = 2$, e a chamada EUCLID(F_3, F_2) faz exatamente uma chamada recursiva, a EUCLID(1, 0). (Temos de começar em $k = 2$ porque, quando $k = 1$ não temos $F_2 > F_1$.) Para o passo de indução, suponha que EUCLID(F_k, F_{k-1}) faz exatamente $k - 2$ chamadas recursivas. Para $k > 2$, temos $F_k > F_{k-1} > 0$ e $F_{k+1} = F_k + F_{k-1}$, e, assim, pelo Exercício 31.1-1, temos $F_{k+1} \bmod F_k = F_{k-1}$. Assim, temos

$$\begin{aligned} \text{mdc}(F_{k+1}, F_k) &= \text{mdc}(F_k, F_{k+1} \bmod F_k) \\ &= \text{mdc}(F_k, F_{k-1}) \end{aligned}$$

Portanto, a chamada EUCLID (F_{k+1}, F_k) executa uma recursão a mais que a chamada EUCLID (F_k, F_{k-1}), ou exatamente $k - 1$ vezes, atingindo o limite superior dado pelo Teorema 31.11.

Visto que F_k é aproximadamente $\varphi^k/\sqrt{5}$, onde f é a razão áurea $(1+\sqrt{5})/2$ definida pela equação (3.24), o número de chamadas recursivas em EUCLID é $O(\lg b)$. (Veja um limite mais preciso no Exercício 31.2-5.) Portanto, se chamarmos EUCLID para dois números de bits, então o procedimento executa $O()$ operações aritméticas e $O(^3)$ operações de bits (supondo que multiplicação e divisão de números de bits demorem $O(^2)$ operações com bits. O Problema 31-2 pede que você mostre um limite $O(^2)$ para o número de operações com bits.

A forma estendida do algoritmo de Euclides

Agora, reescrevemos o algoritmo de Euclides para calcular informações adicionais úteis. Especificamente, estendemos o algoritmo para calcular os coeficientes inteiros x e y tais que

$$d = \text{mdc}(a, b) = ax + by \,. \tag{31.16}$$

Observe que x e y podem ser zero ou negativos. Verificaremos mais adiante que esses coeficientes são úteis para calcular inversos multiplicativos modulares. O procedimento EXTENDED-EUCLID toma como entrada um par de inteiros não negativos e retorna uma tripla da forma (d, x, y) que satisfaz a equação (31.16).

```
EXTENDED-EUCLID(a, b)
1  if b == 0
2        return (a, 1, 0)
3  else (d', x', y') = EXTENDED-EUCLID(b, a mod b)
4        (d, x, y) = (d', y', x' − ⌊a/b⌋ y')
5        return (d, x, y)
```

A Figura 31.1 ilustra o cálculo de mdc(99, 78) por EXTENDED-EUCLID.

O procedimento EXTENDED-EUCLID é uma variação do procedimento EUCLID. A linha 1 é equivalente ao teste "$b == 0$" na linha 1 de EUCLID. Se $b = 0$, então EXTENDED-EUCLID retorna não somente $d = a$ na linha 2, mas também os coeficientes $x = 1$ e $y = 0$, de modo que $a = ax + by$. Se $b \neq 0$, EXTENDED-EUCLID primeiro calcula (d', y', x') tal que $d' = \text{mdc}(b, a \bmod b)$ e

$$d' = bx' + (a \bmod b)\, y' \,. \tag{31.17}$$

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

Figura 31.1 Cálculo de mdc(99,78) por EXTENDED-EUCLID. Cada linha mostra um nível da recursão: os valores das entradas a e b, o valor calculado a/b e os valores d, x e y retornados. A tripla (d, x, y) retornada se torna a tripla (d', x', y') usada no cálculo do próximo nível mais alto de recursão. A chamada EXTENDED-EUCLID(99, 78) retorna (3, -11, 14) e, assim, mdc(99, 78) = 3 = 99 · (-11) + 78 · 14.

Como em EUCLID, temos nesse caso d = mdc(a, b) = d' = mdc(b, a mod b). Para obter x e y tais que $d = ax + by$, começamos reescrevendo a equação (31.17) usando a equação $d = d'$ e a equação (3.8):

$$\begin{aligned} d &= bx' + (a - b\lfloor a/b \rfloor)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y') \,. \end{aligned}$$

Assim, escolher $x = y'$ e $y = x' - \quad a/b \quad y'$ satisfaz a equação $d = ax + by$, provando a correção de EXTENDED-EUCLID.

Visto que o número de chamadas recursivas feitas em EUCLID é igual ao número de chamadas recursivas feitas em EXTENDED-EUCLID, os tempos de execução de EUCLID E EXTENDED-EUCLID são iguais, a menos de um fator constante. Isto é, para $a > b > 0$, o número de chamadas recursivas é $O(\lg b)$.

Exercícios

31.2-1 Prove que as equações (31.11) e (31.12) implicam a equação (31.13).

31.2-2 Calcule os valores (d, x, y) que a chamada EXTENDED-EUCLID(899, 493) retorna.

31.2-3 Prove que, para todos os inteiros a, k e n,

mdc(a, n) = mdc($a + kn$, n).

31.2-4 Reescreva EUCLID em uma forma iterativa que use somente uma quantidade constante de memória (isto é, armazene apenas um número constante de valores inteiros).

31.2-5 Se $a > b \geq 0$, mostre que a chamada EUCLID(a, b) faz no máximo $1 + \log_f b$ chamadas recursivas. Melhore esse limite para $1 + \log_f (b/\text{mdc}(a, b))$.

31.2-6 O que EXTENDED-EUCLID(F_{k+1}, F_k) retorna? Prove a correção da sua resposta.

31.2-7 Defina a função mdc para mais de dois argumentos pela equação recursiva mdc(a_0, a_1, ..., a_n) = mdc(a_0, mdc(a_0, a_1, ..., a_n)). Mostre que a função mdc retorna a mesma resposta, independentemente da ordem na qual seus argumentos são especificados. Mostre também como determinar inteiros x_0, x_1, ..., x_n tais que mdc(a_0, a_1, ..., a_n) = a_0x_0, + a_1x_1, ..., + a_nx_n. Mostre que o número de divisões executadas por seu algoritmo é $O(n + \lg(\max\{a_0, a_1, ..., a_n\}))$.

31.2-8 Defina mmc(a_1, a_2, ..., a_n) como o ***mínimo múltiplo comum*** dos n inteiros a_1, a_2, ..., a_n, isto é, o menor inteiro não negativo que é um múltiplo de cada a_i. Mostre como calcular mmc(a_1, a_2, ..., a_n) eficientemente usando a operação mdc (de dois argumentos) como uma sub-rotina.

31.2-9 Prove que n_1, n_2, n_3 e n_4 são primos dois a dois se e somente se mdc(n_1n_2, n_3,n_4) = mdc (n_1n_3, n_2n_4) = 1.

De modo mais geral, mostre que n_1, n_2, ..., n_k são primos entre si aos pares se e somente se um conjunto de lg k pares de números derivados de n_i são primos entre si.

31.3 ARITMÉTICA MODULAR

Informalmente, podemos entender a aritmética modular como a aritmética usual com inteiros, exceto que, se estamos trabalhando módulo n, todo resultado x é substituído pelo elemento de {0, 1, ..., n – 1}, que é equivalente a x, módulo n (isto é, x é substituído por x mod n). Esse modelo informal é suficiente se nos limitarmos às operações de adição, subtração e multiplicação. Um modelo mais formal para aritmética modular, que damos agora, é mais bem descrito dentro da estrutura da teoria de grupos.

Grupos finitos

Um ***grupo*** (S, $\oplus$) é um conjunto S aliado a uma operação binária $\oplus$ definida em S para a qual são válidas as seguintes propriedades:

1. **Fechamento:** Para todo a, $b \in S$, temos $a \oplus b \in S$.
2. **Identidade:** Existe um elemento $e \in S$, denominado ***identidade*** do grupo, tal que $e \oplus a = a \oplus e = a$ para todo $a \in S$.
3. **Associatividade:** Para todo a, b, $c \in S$, temos $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
4. **Inversos:** Para cada $a \in S$, existe um elemento único $b \in S$, denominado ***inverso*** de a, tal que

$$a \oplus b = b \oplus a = e.$$

Como exemplo, considere o conhecido grupo (, +) dos inteiros na operação de adição: 0 é a identidade, e o inverso de a é $-a$. Se um grupo (S, $\oplus$) satisfaz a ***lei comutativa*** $a \oplus b = b \oplus a$ para todo a, $b \in S$, então ele é um ***grupo abeliano***. Se um grupo (S, $\oplus$) satisfaz $|S| < \infty$, então ele é um ***grupo finito***.

Grupos definidos por adição e multiplicação modulares

Podemos formar dois grupos abelianos finitos usando adição e multiplicação módulo n, onde n é um inteiro positivo. Esses grupos são baseados nas classes de equivalência dos inteiros módulo n, definidas na Seção 31.1.

Para definir um grupo em $_n$, precisamos ter operações binárias adequadas, que obtemos redefinindo as operações comuns de adição e multiplicação. É fácil definir operações de adição e multiplicação para $_n$ porque a classe de equivalência de dois inteiros determina unicamente a classe de equivalência de sua soma ou produto. Isto é, se $a \equiv a'$ (mod n) e $b \equiv b'$ (mod n), então

$$a + b \equiv a' + b' \pmod{n}$$
$$ab \equiv a'b \pmod{n}.$$

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(a)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b)

Figura 31.2 Dois grupos finitos. As classes de equivalência são denotadas por seus elementos representantes. (**a**) O grupo ($_6$, $+_6$). (**b**) O grupo ($^*{}_{15}$, $\cdot_{15}$).

Assim, definimos adição e multiplicação módulo n, denotadas por $+_n$ e $\cdot_n$, por:

$$[a]_n +_n [b]_n = [a + b]_n \, , \tag{31.18}$$
$$[a]_n \cdot_n [b]_n = [ab]_n \, .$$

(Podemos definir, de modo semelhante, a subtração em $_n$ por $[a]_n -_n [b]_n = [a - b]_n$, mas a divisão é mais complicada, como veremos.) Esses fatos justificam a prática comum e conveniente de usar o menor elemento não negativo de cada classe de equivalência como seu representante ao executar cálculos em $_n$. Somamos, subtraímos e multiplicamos como sempre, usando os representantes, mas substituímos cada resultado x pelo representante de sua classe (isto é, por x mod n).

Usando essa definição de adição módulo n, definimos o ***grupo aditivo módulo n*** como ($_n$, $+_n$). O tamanho do grupo aditivo módulo n é $|_n| = n$. A Figura 31.2(a) apresenta a tabela de operação para o grupo ($_6$, $+_6$).

Teorema 31.12

O sistema ($_n$, $+_n$) é um grupo abeliano finito.

Prova A equação (31.18) mostra que ($_n$, $+_n$) é fechado. A associatividade e a comutatividade de $+_n$ decorrem da associatividade e da comutatividade de +:

$$\begin{aligned}
([a]_n +_n [b]_n) +_n [c]_n &= [a+b]_n +_n [c]_n \\
&= [(a+b)+c]_n \\
&= [a+(b+c)]_n \\
&= [a]_n +_n [b+c]_n \\
&= [a]_n +_n ([b]_n + [c]_n)
\end{aligned}$$

$$\begin{aligned}
[a]_n +_n [b]_n &= [a+b]_n \\
&= [b+a]_n \\
&= [b]_n +_n [a]_n \,.
\end{aligned}$$

O elemento identidade de (, $+_n$) é 0 (isto é, $[0]_n$). O inverso (aditivo) de um elemento a (isto é, de $[a]_n$) é o elemento $-a$ (isto é, $[-a]_n$ ou $[n-a]_n$), visto que $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$.

Usando a definição de multiplicação módulo n, definimos o ***grupo multiplicativo módulo n*** como ($^*_n$, $\cdot_n$). Os elementos desse grupo são o conjunto $^*_n$ de elementos em $_n$ que são primos com n:

$$\mathbb{Z}_n^* = \{[a]_n \in \mathbb{Z}_n : \text{mdc}(a,n) = 1\}\,.$$

Para ver que $^*_n$ é bem definido, observe que, para $0 \leq a < n$, temos $a \equiv (a + kn) \pmod n$ para todos os inteiros k. Então, pelo Exercício 31.2-3, $\text{mdc}(a, n) = 1$ implica $\text{mdc}(a + kn, n) = 1$ para todos os inteiros k. Visto que $[a]_n = \{a + kn : k \in \}$, o conjunto $^*_n$. é bem definido. Um exemplo de tal grupo é

$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}\,,$$

onde a operação de grupo é multiplicação módulo 15. (Aqui, denotamos um elemento $[a]^{15}$ por a_{15}; por exemplo, denotamos $[7]^{15}$ por 7.) A Figura 31.2(b) mostra o grupo (^{*15}, $\cdot_{15}$). Por exemplo, $8 \cdot 11 \equiv 13 \pmod{15}$, trabalhando com *. A identidade para esse grupo é 1.

Teorema 31.13

O sistema ($^*_n$, $\cdot_n$) é um grupo abeliano finito.

Prova O Teorema 31.6 implica que ($^*_n$, $\cdot_n$) é fechado. Associatividade e comutatividade podem ser provadas para $\cdot_n$ como foram para $+_n$ na prova do Teorema 31.12. O elemento identidade é $[1]_n$ Para mostrar a existência de inversos, seja a um elemento de ($^*_n$), e seja (d, x, y) retornado por EXTENDED-EUCLID(a, n). Então $d = 1$, já que $a \in {}^*_n$, e

$$ax + ny = 1 \tag{31.19}$$

ou, o que é equivalente,

$$ax \equiv 1 \pmod n\,.$$

Assim, $[x]_n$ é um inverso multiplicativo de $[a]_n$, módulo n. Além disso, afirmamos que $[x]_n \in {}_n$. Para ver por que, a equação (31.19) demonstra que a menor combinação linear possível de x e n deve ser 1. Portanto, o Teorema 31.2 implica que $\text{mdc}(x, n) = 1$. Adiamos para o Corolário 31.26, a prova de que inversos são unicamente definidos.

Como exemplo do cálculo de inversos multiplicativos, suponha que $a = 5$ e $n = 11$. Então, EXTENDED-EUCLID(a, n) retorna $(d, x, y) = (1, –2, 1)$, de modo que $1 = 5 \cdot (–2) + 11 \cdot 1$. Assim, $[–2]_{11}$ (isto é, $[9]_{11}$) é o inverso multiplicativo de $[5]_{11}$.

Quando trabalharmos com os grupos ($_n$, $+_n$) e ($^*_n$, $\cdot_n$) no restante deste capítulo, seguiremos a prática conveniente de denotar classes de equivalência por seus elementos representantes e denotar as operações $+_n$ e $\cdot_n$ pelas notações aritméticas usuais + e · (ou justaposição, de modo que $ab = a \cdot b$), respectivamente. Além disso, equivalências módulo n também podem ser interpretadas como equações em $_n$. Por exemplo, as duas declarações seguintes são equivalentes:

$$ax \equiv b \pmod{n}\ .$$

$$[a]_n \cdot_n [x]_n = [b]_n\ .$$

Como conveniência adicional, às vezes, nos referimos a um grupo (S, ⊕) somente como S quando a operação é entendida pelo contexto. Assim, podemos nos referir aos grupos ($_n$, $+_n$) e ($^*{}_n$, $\cdot_n$) como $_n$ e * , respectivamente.

Denotamos o inverso (multiplicativo) de um elemento a por (a^{-1} mod n). Divisão em * . é definida pela equação $a / b \equiv ab^{-1}$. Por exemplo, em $_{15}$ temos que $7^{-1} \equiv 13 \pmod{15}$, visto que $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$, de modo que $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

O tamanho de $^*{}_n$ é denotado por $f(n)$. Essa função, conhecida como ***função fi de Euler***, satisfaz a equação

$$\phi(n) = n \prod_{p\,:\,p \text{ é primo e } p|n} \left(1 - \frac{1}{p}\right) \tag{31.20}$$

de modo que p percorre todos os primos que dividem n (inclusive o próprio n, se n é primo). Não provaremos essa fórmula aqui. Intuitivamente, começamos com uma lista dos n restos $\{0, 1, ..., n – 1\}$ e depois, para cada primo p que divide n, cancelamos todos os múltiplos de p na lista. Por exemplo, como os divisores primos de 45 são 3 e 5,

$$\begin{aligned} \phi(45) &= 45\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right) \\ &= 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) \\ &= 24 \end{aligned}$$

Se p é primo, então $^* = \{1, 2, ..., p – 1\}$ e

$$\begin{aligned} \phi(p) &= p\left(1-\frac{1}{p}\right) \\ &= p-1 \end{aligned} \tag{31.21}$$

Se n é composto, então $f(n) < n – 1$, mas pode-se mostrar que

$$\phi(n) > \frac{n}{e^{\gamma} \ln\ln n + \frac{3}{\ln\ln n}} \tag{31.22}$$

para $n \geq 3$, onde $= 0{,}5772156649...$ é a ***constante de Euler***. Um limite inferior um pouco mais simples (porém mais relaxado) para $n > 5$ é

$$\phi(n) > \frac{n}{6 \ln\ln n} \tag{31.23}$$

O limite inferior (31.22) é essencialmente o melhor possível, visto que

$$\liminf_{n\to\infty} \frac{\phi(n)}{n/\ln\ln n} = e^{-\gamma} \tag{31.24}$$

Subgrupos

Se (S, ⊕) é um grupo, $S' \subseteq S$ e (S', ⊕) também é um grupo, então (S', ⊕) é um ***subgrupo*** de (S, ⊕). Por exemplo, os inteiros pares formam um subgrupo dos inteiros para a operação de adição. O teorema a seguir dá uma ferramenta útil para reconhecer subgrupos.

Teorema 31.14 (Um subconjunto fechado não vazio de um grupo finito é um subgrupo)

Se (S, ⊕) é um grupo finito e S' é qualquer subconjunto não vazio de S tal que $a \oplus b \in S'$ para todo a, $b \in S'$, então (S', ⊕) é um subgrupo de (S, ⊕).

Prova Deixamos a prova para o Exercício 31.3-3.

Por exemplo, o conjunto {0, 2, 4, 6} forma um subgrupo de ^{8}, já que é não vazio e fechado pela operação + (isto é, é fechado para $+_8$).

O teorema a seguir dá uma restrição extremamente útil para o tamanho de um subgrupo; omitimos a prova.

Teorema 31.15 (Teorema de Lagrange))

Se (S, ⊕) é um grupo finito e (S', ⊕) é um subgrupo de (S, ⊕), então $|S'|$ é um divisor de $|S|$.

Um subgrupo S' de um grupo S é um subgrupo ***próprio*** se $S' \neq S$. Usaremos o corolário a seguir na análise que fazemos na Seção 31.8 do procedimento de teste de primalidade de Miller-Rabin.

Corolário 31.16

Se S' é um subgrupo próprio de um grupo finito S, então $|S'| \leq |S|/2$.

Subgrupos gerados por um elemento

O Teorema 31.14 nos dá um modo fácil para produzir um subgrupo de um grupo finito (S, ⊕): escolha um elemento a e tome todos os elementos que podem ser gerados de a usando a operação de grupo. Especificamente, defina $a_{(k)}$ para $k \geq 1$ por

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \ldots \oplus a}_{k}$$

Por exemplo, se tomarmos $a = 2$ no grupo ^{6}, a sequência $a_{(1)}$, $a_{(2)}$, ... é 2, 4, 0, 2, 4, 0, 2, 4, 0, ...

No grupo $_n$, temos $a_{(k)} = ka \bmod n$, e no grupo *, temos $a\ (^{k}) = a_k \bmod n$. Definimos o ***sub-grupo gerado por a***, denotado por $\langle a \rangle$ ou ($\langle a \rangle$, ⊕), por

$$\langle a \rangle = \{a^{(k)} : k \geq 1\} .$$

Dizemos que a ***gera*** o subgrupo $\langle a \rangle$, ou que a é um ***gerador*** de $\langle a \rangle$. Visto que S é finito, $\langle a \rangle$ é um subconjunto finito de S, possivelmente incluindo todo o conjunto S. Como a associatividade de $\oplus$ implica

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)} ,$$

$\langle a \rangle$ é fechado e, portanto, pelo Teorema 31.14, $\langle a \rangle$ é um subgrupo de S. Por exemplo, em $_6$, temos

$$\langle 0 \rangle = \{0\} ,$$
$$\langle 1 \rangle = \{0, 1, 2, 3, 4, 5\} ,$$
$$\langle 2 \rangle = \{0, 2, 4\} .$$

De modo semelhante, em * , temos

$$\langle 1 \rangle = \{1\} ,$$
$$\langle 2 \rangle = \{1, 2, 4\} ,$$
$$\langle 3 \rangle = \{1, 2, 3, 4, 5, 6\} .$$

A ***ordem*** de a (no grupo S), denotada por ord(a), é definida como o menor inteiro positivo t tal que $a^{(t)} = e$.

Teorema 31.17

Para qualquer grupo finito (S, $\oplus$) e qualquer $a \in S$, a ordem de um elemento é igual ao tamanho do subgrupo que ele gera ou ord(a) = $|\langle a \rangle|$.

Prova Seja t = ord(a). Visto que $a^{(t)} = e$ e $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ para $k \geq 1$, se $i > t$, então $a^{(i)} = a^{(j)}$ para algum $j < i$. Assim, à medida que geramos elementos por a, não vemos nenhum elemento depois de $a^{(t)}$. Portanto, $\langle a \rangle = \{a_{(1)}, a_{(2)}, ..., a^{(t)}\}$ e $|\langle a \rangle| \leq t$. Para mostrar que $|\langle a \rangle| \geq t$, mostramos que cada elemento da sequência $a_{(1)}, a_{(2)}, ..., a^{(t)}$ é distinto. Suponha, por contradição, que $a_{(i)} = a_{(j)}$ para algum i e j que satisfaça $1 \leq i < j \leq t$. Então, $a^{(i+k)} = a^{(j+k)}$ para $k \geq 0$. Mas essa igualdade implica que $a^{(i+(t-j))} = a^{(j+(t-j))} = e$, uma contradição, já que $i + (t - j) < t$, mas t é o menor valor positivo tal que $a_{(t)} = e$. Portanto, cada elemento da sequência $a_{(1)}, a_{(2)}, ..., a_{(t)}$ é distinto, e $|\langle a \rangle| \geq t$. Concluímos que ord($a$) = $|\langle a \rangle|$.

Corolário 31.18

A sequência $a_{(1)}, a_{(2)}, ...$ é periódica com período t = ord(a); isto é, $a^{(i)} = a^{(j)}$ se e somente se $i \equiv j$ (mod t).

Compativelmente com o Corolário 31.18, definimos $a_{(0)}$ como e e $a^{(i)}$ como $a^{(i \bmod t)}$, onde t = ord(a), para todos os inteiros i.

Corolário 31.19

Se (S, $\oplus$) é um grupo finito com identidade e, então para todo $a \in S$,

$$a^{(|S|)} = e .$$

Prova O teorema de Lagrange (Teorema 31.15) implica que ord(a) | $|S|$, e portanto $|S| \equiv 0$ (mod t), onde t = ord(a). Então, $a_{(|S|)} = a_{(0)} = e$.

31.3-1 Desenhe as tabelas de operação de grupo para os grupos (4, +4) e (*5, ·$_5$) . Mostre que esses grupos são isomorfos, exibindo uma correspondência de um para um α entre seus elementos, tal que $a + b \equiv c$ (mod 4) se e somente se a(a) · $\alpha(b) \equiv \alpha(c)$ (mod 5).

31.3-2 Organize uma lista com todos os subgrupos de 9 e de *13.

31.3-3 Prove o Teorema 31.14.

31.3-4 Mostre que, se p é primo e e é um inteiro positivo, então

$$\phi(p^e) = p^{e-1}(p-1) .$$

31.3-5 Mostre que, para qualquer inteiro $n > 1$ e para qualquer $a \in$ *, a função f_a: * $_n$ → *$_n$ definida por $f_a(x) = ax$ mod n é uma permutação de *$_n$.

31.4 Solução de equações lineares modulares

Agora, consideramos o problema de encontrar soluções para a equação

$$ax \equiv b \pmod{n} , \qquad (31.25)$$

onde $a > 0$ e $n > 0$. Esse problema tem várias aplicações; por exemplo, nós o usaremos como parte do procedimento para determinar chaves no sistema de criptografia de chave pública RSA, na Seção 31.7. Supomos que a, b e n são dados e desejamos determinar todos os valores de x, módulo n, que satisfazem a equação (31.25). A equação pode ter zero, uma ou mais de uma solução como essa.

Seja $\langle a \rangle$ o subgrupo de $_n$ gerado por a. Visto que $\langle a \rangle = \{a_{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, a equação (31.25) tem uma solução se e somente se $b \in \langle a \rangle$. O teorema de Lagrange (Teorema 31.15) afirma que $|\langle a \rangle|$ deve ser um divisor de n. O teorema a seguir nos dá uma caracterização precisa de $\langle a \rangle$.

Teorema 31.20

Para quaisquer inteiros positivos a e n, se $d = \text{mdc}(a, n)$, então

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, ..., ((n/d) - 1)d\} , \qquad (31.26)$$

em $_n$ e, assim,

$$|\langle a \rangle| = n/d .$$

Prova Começamos mostrando que $d \in \langle a \rangle$. Lembre-se de que EXTENDED-EUCLID(a, n) produz inteiros x' e y' tais que $ax' + ny' = d$. Assim, $ax' \equiv d$, de modo que $d \in \langle a \rangle$. Em outras palavras, d é múltiplo de a em $_n$.

Visto que $d \in \langle a \rangle$, decorre que todo múltiplo de d pertence a $\langle a \rangle$ porque qualquer múltiplo de um múltiplo de a é ele próprio um múltiplo de a. Então, $\langle a \rangle$ contém todos os elementos em $\{0, d, 2d, ..., ((n/d) - 1)d\}$. Isto é, $\langle d \rangle \subseteq \langle a \rangle$.

Agora mostramos que $\langle a \rangle \subseteq \langle d \rangle$. Se $m \in \langle a \rangle$, então $m = ax$ mod n para algum inteiro x, e portanto $m = ax + ny$ para algum inteiro y. Contudo, $d \mid a$ e $d \mid n$, e assim $d \mid m$ pela equação (31.4). Portanto, $m \in \langle d \rangle$.

Combinando esses resultados, temos que $\langle a \rangle = \langle d \rangle$. Para ver que $|\langle a \rangle| = n/d$, observe que há exatamente n/d múltiplos de d entre 0 e $n - 1$, inclusive.

Corolário 31.21

A equação $ax \equiv b$ pode ser resolvida para a incógnita x se e somente se $d|\ b$, onde $d = \text{mdc}(a, n)$.

Prova A equação $ax \equiv b \pmod{n}$ pode ser resolvida se e somente se $[b] \in \langle a \rangle$, que é o mesmo que dizer .

$$(b \bmod n) \in \{0, d, 2d, ..., (n/d) - 1)d\}$$

pelo Teorema 31.20. Se $0 \le b < n$, então $b \in \langle a \rangle$ se e somente se $d \mid b$, visto que os membros de $\langle a \rangle$ são exatamente os múltiplos de d. Se $b < 0$ ou $b \ge$ n, então o corolário decorre da observação de que $d \mid b$ se e somente se $d \mid (b \bmod n)$, já que a diferença entre b e $b \bmod n$ é um múltiplo de n, que é, ele próprio, um múltiplo de d.

Corolário 31.22

A equação $ax \equiv b \pmod{n}$ tem d soluções distintas módulo n, onde $d = \text{mdc}(a, n)$ ou não tem nenhuma solução.

Prova Se $ax \equiv b$ (mod n) tem uma solução, então $b \in \langle a \rangle$. Pelo Teorema 31.17, ord(a) = $|\langle a \rangle|$ e, assim, o Corolário 31.18 e o Teorema 31.20 implicam que a sequência $ai \bmod n$, para $i = 0, 1, ...$ é periódica com período $|\langle a \rangle| = n/d$. Se $b \in \langle a \rangle$, então b aparece exatamente d vezes na sequência $ai \bmod n$, para $i = 0, 1, ..., n - 1$, já que o bloco de valores $\langle a \rangle$ de comprimento (n/d) é repetido exatamente d vezes à medida que i cresce de 0 a $n - 1$. Os índices x das d posições para as quais $ax = \bmod n = b$ são as soluções da equação $ax \equiv b \pmod{n}$.

Teorema 31.23

Seja $d = \text{mdc}(a, n)$ e suponha que $d = ax' + ny'$ para alguns inteiros x' e y' (por exemplo, como calculado por EXTENDED-EUCLID). Se $d \mid b$, então a equação $ax \equiv b \pmod{n}$ tem como uma de suas soluções o valor x_0, onde

$$x_0 = x'(b/d) \bmod n\ .$$

Prova Temos

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod{n} \\ &\equiv d(b/d) \pmod{n} \qquad (\text{porque } ax' \equiv d \pmod{n}) \\ &\equiv b \pmod{n}, \end{aligned}$$

e assim x_0 é uma solução para $ax \equiv b \pmod{n}$.

Teorema 31.24

Suponha que a equação $ax \equiv b \pmod{n}$ tenha solução (isto é, $d \mid b$, onde $d = \text{mdc}(a, n)$), e que x_0 seja alguma solução para essa equação. Então, essa equação tem exatamente d soluções distintas, módulo n, dadas por $x_i = x_0 + i(n/d)$ para $i = 0, 1, ..., d - 1$.

Prova Como $n/d > 0$ e $0 \le i(n/d) < n$ para $i = 0, 1, ..., d - 1$, os valores $x_0, x_1, ..., x_{d-1}$ são todos distintos, módulo n. Como x_0 é uma solução de $ax \equiv b \pmod{n}$, temos $ax_0 \bmod n \equiv b \pmod{n}$. Assim, para $i = 0, 1, ..., d - 1$, temos

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n \\ &= a(x_0 + ain/d) \bmod n \\ &= ax_0 \bmod n \qquad (\text{porque } d \mid a \text{ implica que } a\,i\,n/d \text{ é um múltiplo de } n) \\ &= b \pmod{n}, \end{aligned}$$

e, por consequência, $ax_i \equiv b \pmod{n}$, o que faz de x_i uma solução também. Pelo Corolário 31.22, a equação a $ax \equiv b \pmod{n}$ tem exatamente d soluções, de modo que $x_0, x_1, ..., x_{d-1}$ devem ser todas elas.

Agora, desenvolvemos a matemática necessária para resolver a equação $ax \equiv b \pmod{n}$; o algoritmo a seguir imprime todas as soluções para essa equação. As entradas a e n são inteiros positivos arbitrários e b é um inteiro arbitrário.

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') = EXTENDED-EUCLID(a, n)
2  if d | b
3        x0 = x'(b/d) mod n
4        for i = 0 to d − 1
5             imprima (x0 + i(n/d)) mod n
6  else imprimir "nenhuma solução"
```

Como exemplo da operação desse procedimento, considere a equação $14x \equiv 30 \pmod{100}$ (aqui, $a = 14$, $b = 30$ e $n = 100$). Chamando EXTENDED-EUCLID na linha 1, obtemos $(d, x, y) = (2, -7, 1)$. Como $2 \mid 30$, as linhas 3–5 são executadas. A linha 3 calcula $x_0 = (-7)(15) \bmod 100 = 95$. O laço nas linhas 4–5 imprime as duas soluções: 95 e 45.

O procedimento MODULAR-LINEAR-EQUATION-SOLVER funciona da seguinte maneira: a linha 1 calcula $d = \text{mdc}(a, n)$ bem como dois valores x' e y' tais que $d = ax' + ny'$, demonstrando que x' é uma solução para a equação $ax' \equiv d \pmod{n}$. Se d não divide b, então a equação $ax \equiv b \pmod{n}$ não tem nenhuma solução, pelo Corolário 31.21. A linha 2 verifica se $d \mid b$; se não, a linha 6 informa que não existe nenhuma solução. Caso contrário, a linha 3 calcula uma solução x_0 para $ax \equiv b \pmod{n}$, de acordo com o Teorema 31.23. Dada uma solução, o Teorema 31.24 informa que somar múltiplos de (n/d), módulo n, produz as outras $d - 1$ soluções. O laço **for** das linhas 4–5 imprime todas as d soluções, começando com x_0 e espaçadas entre si (n/d), módulo n.

MODULAR-LINEAR-EQUATION-SOLVER executa $O(\lg n + \text{mdc}(a, n))$ operações aritméticas, já que EXTENDED-EUCLID executa $O(\lg n)$ operações aritméticas, e cada iteração do laço **for** das linhas 4–5 executa um número constante de operações aritméticas.

Os seguintes corolários do Teorema 31.24 dão especializações de particularmente interessantes.

Corolário 31.25

Para qualquer $n > 1$, se $\text{mdc}(a, n) = 1$, então a equação $ax \equiv b \pmod{n}$ tem uma solução única, módulo n.

Se $b = 1$, um caso comum de interesse considerável, o x que estamos procurando é um ***inverso multiplicativo*** de a, módulo n.

Corolário 31.26

Para qualquer $n > 1$, se $\text{mdc}(a, n) = 1$, então a equação $ax \equiv 1 \pmod{n}$ tem uma solução única, módulo n. Caso contrário, ela não tem nenhuma solução.

Graças ao Corolário 31.26 podemos usar a notação $a_{-1} \bmod n$ para nos referirmos ao inverso multiplicativo de a, módulo n, quando a e n são primos entre si. Se $\text{mdc}(a, n) = 1$, então a única solução para a equação $ax \equiv 1 \pmod{n}$ é o inteiro x retornado por EXTENDED-EUCLID, já que a equação

$$\text{mdc}(a, n) = 1 = ax + ny$$

implica $ax \equiv 1 \pmod{n}$. Assim, podemos calcular $a^{-1} \bmod n$ eficientemente usando EXTENDED-EUCLID.

31.4-1 Determine todas as soluções para a equação $35x \equiv 10 \pmod{50}$.

31.4-2 Prove que a equação $ax \equiv ay \pmod{n}$ implica $x \equiv y \pmod{n}$ sempre que $\text{mdc}(a, n) = 1$. Mostre que a condição $\text{mdc}(a, n) = 1$ é necessária, fornecendo um contraexemplo com $\text{mdc}(a, n) > 1$.

31.4-3 Considere a seguinte mudança na linha 3 do procedimento MODULAR-LINEAR-EQUA- TION-SOLVER:

$$3 \quad x_0 = x'(b/d) \bmod (n/d)$$

Isso funcionará? Explique sua resposta.

31.4-4 ★ Seja p primo e $f(x) \equiv f_0 + f_1x + ... + f_tx^t \pmod{p}$ um polinômio de grau t, com coeficientes f_i extraídos de $_p$. Dizemos que a $_p$ ∈ é um ***zero*** de f se $f(a) \equiv 0 \pmod{p}$. Prove que, se a é um zero de f, então $f(x) \equiv (x - a)\, g(x) \pmod{p}$ para algum polinômio $g(x)$ de grau $t - 1$. Prove por indução em relação a t que, se p é primo, então um polinômio $f(x)$ de grau t pode ter no máximo t zeros distintos módulo p.

31.5 O TEOREMA CHINÊS DO RESTO

Por volta do ano 100, o matemático chinês Sun-Tsu resolveu o problema de determinar os inteiros x que deixam restos 2, 3 e 2 quando divididos por 3, 5 e 7, respectivamente. Uma dessas soluções é $x = 23$; todas as soluções têm a forma $23 + 105k$ para inteiros arbitrários k. O "teorema chinês do resto" dá uma correspondência entre um sistema de equações módulo de um conjunto de primos dois a dois (por exemplo, 3, 5 e 7) e uma equação módulo, o produto desses primos (por exemplo, 105).

O teorema chinês do resto tem duas aplicações importantes. Considere o inteiro n fatorado como $n = n_1n_2 \cdots n_k$, onde os fatores n_i são primos entre si aos pares. Em primeiro lugar, o teorema chinês do resto é um "teorema de estrutura" descritivo que mostra a estrutura de $_n$ como idêntica à do produto cartesiano a linha base do x tem que ser a mesma do $_{n^1} \times {}_{n^2} \times ... \times {}_{nk}$, com adição e multiplicação componente a componente módulo n_i na i-ésima componente. Em segundo lugar, essa descrição nos ajuda a projetar algoritmos eficientes, já que trabalhar em cada um dos sistemas $_{ni}$ pode ser mais eficiente (em termos de operações com bits) que trabalhar com módulo n.

Teorema 31.27 (Teorema chinês do resto)

Seja $n = n_1n_2 \cdots n_k$, onde os n_i são primos dois a dois. Considere a correspondência

$$a \leftrightarrow (a_1, a_2, ..., a_k) \tag{31.27}$$

onde $a \in {}_n$, $a_i \in {}_{n^i}$, e

$$a_i = a \bmod n_i$$

para $i = 1, 2, ..., k$. Então, mapear (31.27) é uma correspondência biunívoca (bijeção) entre $_n$ e o produto cartesiano $_{n^1} \times {}_{n^2} \times ... \times {}_{nk}$. Operações executadas nos elementos de $_n$ podem ser executadas de modo equivalente nas k tuplas correspondentes se as operações forem executadas independentemente em cada posição coordenada no sistema adequado. Isto é, se

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) \ ,$$

$$b \leftrightarrow (b_1, b_2, \ldots, b_k) \ ,$$

então

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k) \ , \quad (31.28)$$

$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k) \ , \quad (31.29)$$

$$(ab) \bmod n \leftrightarrow ((a_1 b_1 \bmod n_1, \ldots, a_k b_k) \bmod n_k) \ . \quad (31.30)$$

Prova Transformar uma representação na outra é razoavelmente simples. Passar de a para $(a_1, a_2, \ldots, a_k)$ é bem fácil e requer somente k operações "mod".

Calcular a pelas entradas $(a_1, a_2, \ldots, a_k)$ é um pouco mais complicado. Começamos definindo $m_i = n/n_i$ para $i = 1, 2, \ldots, k$; assim, m_i é o produto de todos os n_j exceto n_i: $m_i = n_1 n_2 \cdots n_{i-1} n_{i+1} \cdots n_k$. Em seguida, definimos

$$c_i = m_i \cdot (m_i^{-1} \bmod n_i) \quad (31.31)$$

para $i = 1, 2, \ldots, k$. A equação 31.31 é sempre bem definida: visto que m_i e n_i são primos entre si (pelo Teorema 31.6), o Corolário 31.26 garante que $m_i^{-1} \pmod{n_i}$ existe. Finalmente, podemos calcular a como uma função de $a_1, a_2, \ldots, a_k$ assim:

$$a \equiv (a_1 c_1 + a_2 c_2 + \ldots + a_k c_k) \pmod{n}. \quad (31.32)$$

Agora mostramos que a equação (31.32) garante que $a \equiv a_i \pmod{n_i}$ para $i = 1, 2, \ldots, k$. Observe que, se $j \neq i$, então $m_j \equiv 0 \pmod{n_i}$, o que implica que $c_j \equiv m_j \equiv 0 \pmod{n_i}$. Observe também que $c_j \equiv 1 \pmod{n_i}$, pela equação (31.31). Assim, temos a interessante e útil correspondência

$$c_j \leftrightarrow (0, 0, \ldots, 0, 1, 0, \ldots, 0),$$

um vetor que tem zeros em todos os lugares, exceto na i-ésima coordenada, onde tem um 1; assim, em certo sentido, as c_i formam uma "base" para a representação. Então, para cada i, temos

$$\begin{aligned} a &\equiv a_i c_i && \pmod{n_i} \\ &\equiv a_i m_i \, (m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &\equiv a_i && \pmod{n_i}, \end{aligned}$$

que é o que queríamos demonstrar: nosso método de calcular a pelos a_i produz um resultado a que satisfaz as restrições $a \equiv a_i \pmod{n_i}$ para $i = 1, 2, \ldots, k$. A correspondência é biunívoca, já que podemos transformar em ambas as direções. Finalmente, as equações (31.28)–(31.30) decorrem diretamente do Exercício 31.1-7, já que $x \bmod n_i = (x \bmod n) \bmod n_i$ para qualquer x e $i = 1, 2, \ldots, k$.

Os corolários a seguir serão usados mais adiante neste capítulo.

Corolário 31.28

Se $n_1, n_2, \ldots, n_k$ são primos dois a dois e $n = n_1, n_2, \ldots, n_k$ então, para quaisquer inteiros $a_1, a_2, \ldots, a_k$, o conjunto de equações simultâneas

$$x \equiv a_i \pmod{n_i} ,$$

para $i = 1, 2, ..., k$, tem uma solução única módulo n para a incógnita x.

Corolário 31.29

Se $n_1, n_2, ..., n_k$ são primos dois a dois e $n = n_1, n_2, ..., n_k$, então, para todos os inteiros x e a,

$$x \equiv a \pmod{n_i} ,$$

para $i = 1, 2, ..., k$ se e somente se

$$x \equiv a \pmod{n} .$$

Como exemplo da aplicação do teorema chinês do resto, suponha que temos as duas equações

$$a \equiv 2 \pmod{5} ,$$

$$a \equiv 3 \pmod{13} ,$$

de modo que $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$ e $n_2 = m_1 = 13$, e desejamos calcular a mod 65, já que $n = 65$. Como $13^{-1} \equiv 2 \pmod{5}$ e $5^{-1} \equiv 8 \pmod{13}$, temos

$$c_1 \equiv 13 \cdot 2 = 26 ,$$
$$c_2 \equiv 5 \cdot 8 = 40 ,$$

e

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 && \pmod{65} \\ &\equiv 52 + 120 && \pmod{65} \\ &\equiv 42 && \pmod{65} . \end{aligned}$$

A Figura 31.3 apresenta uma ilustração do teorema chinês do resto, módulo 65.

Assim, podemos trabalhar em módulo n trabalhando diretamente em módulo n ou na representação transformada usando cálculos módulo n_i separados, como for conveniente. Os cálculos são totalmente equivalentes.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	29
1	26	1	41	16	56	31	6	46	29	61	36	11	51
2	52	27	2	42	17	57	32	7	47	26	62	37	12
3	13	53	28	3	43	18	58	33	8	48	27	63	38
4	39	14	54	29	4	44	20	59	34	9	49	28	64

Figura 31.3 Ilustração do teorema chinês do resto para $n_1 = 5$ e $n_2 = 13$. Nesse exemplo, $c_1 = 26$ e $c_2 = 40$. Na linha i, coluna j é mostrado o valor de a, módulo 65, tal que a mod $5 = i$ e a mod $13 = j$. Observe que a linha 0, coluna 0 contém um 0. De modo semelhante, a linha 4, coluna 12 contém um 64 (equivalente a -1). Visto que $c_1 = 26$, descer uma linha aumenta a de 26. De modo semelhante, $c_2 = 40$ significa que passar para uma coluna à direita aumenta a de 40. Aumentar a de 1 corresponde a descer pela diagonal para a direita, retornar de baixo para cima e da direita para a esquerda.

Exercícios

31.5-1 Determine todas as soluções para as equações $x \equiv 4 \pmod 5$ e $x \equiv 5 \pmod{11}$.

31.5-2 Determine todos os inteiros x que deixam restos 1, 2, 3 quando divididos por 9, 8, 7, respectivamente.

31.5-3 Demonstre que, de acordo com as definições do Teorema 31.27, se mdc(a, n) = 1, então

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \ldots, (a_k^{-1} \bmod n_K)) .$$

31.5-4 Pelas definições do Teorema 31.27, prove que, para qualquer polinômio f, o número de raízes da equação $f(x) \equiv 0 \pmod n$ é igual ao produto do número de raízes de cada uma das equações $f(x) \equiv 0 \pmod{n_1}$, $f(x) \equiv 0 \pmod{n_2}$, ..., $f(x) \equiv 0 \pmod{n_k}$.

31.6 POTÊNCIAS DE UM ELEMENTO

Exatamente como muitas vezes consideramos os múltiplos de dado elemento a, módulo n, consideramos a sequência de potências de a, módulo n, onde $a \in {}^*_n$:

$$a^0, a^1, a^2, a^3, \ldots, \qquad (31.33)$$

módulo n. Iniciando a indexação de 0, o 0-ésimo valor nessa sequência é a_0 mod n = 1 e o i-ésimo valor é a_i mod n. Por exemplo, as potências de 3 módulo 7 são

i	0	1	2	3	4	5	6	7	8	9	10	11	...
3^i mod 7	1	3	2	6	4	5	1	3	2	6	4	5	...

enquanto as potências de 2 módulo 7 são

i	0	1	2	3	4	5	6	7	8	9	10	11	...
2^i mod 7	1	2	4	1	2	4	1	2	4	1	2	4	...

Nesta seção, representamos por $\langle a \rangle$ o subgrupo de ${}^*_n$ gerado por a por multiplicação repetida, e representamos por ord (a) (a "ordem de a, módulo n") a ordem de a em ${}^*_n$. Por exemplo, $\langle 2 \rangle = \{1, 2, 4\}$ em ^{*7}, e ord^7(2) = 3. Usando a definição da função *fi* de Euler $f(n)$ como o tamanho de ${}^*_n$ (consulte a Seção 31.3), agora traduzimos o Corolário 31.19 para a notação de ${}^*_n$, para obter o teorema de Euler e especializá-lo para ${}^*_n$, onde p é primo, para obter o teorema de Fermat.

Teorema 31.30 (Teorema de Euler)

Para qualquer inteiro $n > 1$,

$$a^{\phi(n)} \equiv 1 \pmod n \text{ para todo } a \in \mathbb{Z}_n^*.$$

Teorema 31.31 (Teorema de Fermat)

Se p é primo, então

$$a^{p-1} \equiv 1 \text{ para todo } a \in \mathbb{Z}_p^* .$$

Prova Pela equação (31.21), $f(n) = p - 1$ se p é primo.

O teorema de Fermat se aplica a todo elemento em $_p$ exceto 0, visto que $0 \notin {}^*_p$. Contudo, para todo $a \in {}_p$, temos $a_p \equiv a \pmod{p}$ se p é primo.

Se $\text{ord}_n(g) = |^*_n|$, então todo elemento em $^*_n$ é uma potência de g, módulo n, e g é uma ***raiz primitiva*** ou um ***gerador*** de $^*_n$. Por exemplo, 3 é uma raiz primitiva, módulo 7, mas 2 não é uma raiz primitiva, módulo 7. Se $^*_n$. possui uma raiz primitiva, o grupo $^*_n$ é ***cíclico***. Omitimos a prova do teorema a seguir, que é provado por Niven e Zuckerman [265].

Teorema 31.32

Os valores de $n > 1$ para os quais $^*_n$ é cíclico são 2, 4, p_e e $2p_e$, para todos os primos $p > 2$ e todos os inteiros positivos e.

Se g é uma raiz primitiva de $^*_n$ e a é qualquer elemento de $^*_n$, então existe um z tal que $g_z \equiv a \pmod{n}$. Esse z é um ***logaritmo discreto*** ou um ***índice*** de a, módulo n, na base g; denotamos esse valor como $\text{ind}_{n}, g(a)$.

Teorema 31.33 (Teorema do logaritmo discreto)

Se g é uma raiz primitiva de $^*_n$, então a equação $g_x \equiv g_y \pmod{n}$ é válida se e somente se a equação $x \equiv y \pmod{f(n)}$ é válida.

Prova Primeiro, suponha que $x \equiv y \pmod{f(n)}$. Então, $x = y + kf(n)$ para algum inteiro k. Assim,

$$\begin{aligned} g^x &\equiv g^{x+k\phi(n)} && \pmod{n} \\ &\equiv g^y \left(g^{\phi(n)}\right)^k && \pmod{n} \\ &\equiv g^y \cdot 1^k && \pmod{n} \quad \text{(pelo teorema de Euler)} \\ &\equiv g^y && \pmod{n} \end{aligned}$$

Ao contrário, suponha que $g_x \equiv g_y \pmod{n}$. Como a sequência de potências de g gera todo elemento de $\langle g\rangle$ e $|\langle g\rangle| = f(n)$, o Corolário 31.18 implica que a sequência de potências de g é periódica com período $f(n)$. Portanto, se $g_x \equiv g_y \pmod{n}$, então devemos ter $x \equiv y \pmod{f(n)}$.

Agora, voltamos nossa atenção às raízes quadradas de 1, módulo potência prima. O teorema a seguir será útil no desenvolvimento de um teste de primalidade do algoritmo na Seção 31.8.

Teorema 31.34

Se p é um primo ímpar e $e \geq 1$, então a equação

$$x^2 \equiv 1 \pmod{p^e} \qquad (31.34)$$

tem somente duas soluções, isto é, $x = 1$ e $x = -1$.

Prova A equação (31.34) é equivalente a

$$p^e \mid (x-1)(x+1).$$

Visto que $p > 2$, podemos ter $p \mid (x - 1)$ ou $p \mid (x + 1)$, mas não ambos. (Caso contrário, pela propriedade (31.3), p também dividiria a diferença entre eles, $(x + 1) - (x - 1) = 2$.) Se p $(x - 1)$, então mdc $(p_e, x - 1) = 1$ e, pelo

Corolário 31.5, teríamos $p_e \mid (x + 1)$. Isto é, $x \equiv -1 \pmod{p_e}$. Simetricamente, se p $(x + 1)$, então mdc $(p_e, x + 1) = 1$, e o Corolário 31.5 implica que $p_e \mid (x - 1) = 1$, de modo que $x \equiv -1 \pmod{p_e}$. Portanto, $x \equiv -1 \pmod{p_e}$ ou $x \equiv 1 \pmod{p_e}$.

Um número x é uma ***raiz quadrada não trivial de 1*, *módulo n***, se satisfaz a equação $x_2 \equiv 1 \pmod{n}$, mas x não é equivalente a nenhuma das duas raízes quadradas "triviais": 1 ou –1, módulo n. Por exemplo, 6 é uma raiz quadrada não trivial de 1, módulo 35. Usaremos o corolário do Teorema 31.34 apresentado a seguir na prova de correção do procedimento de teste de primalidade de Miller-Rabin na Seção 31.8.

Corolário 31.35

Se existe uma raiz quadrada não trivial de 1, módulo n, então n é composto.

Prova Pelo contrapositivo do Teorema 31.34, se existe uma raiz quadrada não trivial de 1, módulo n, então n não pode ser um primo ímpar ou uma potência de um primo ímpar. Se $x_2 \equiv 1 \pmod{2}$, então $x \equiv 1 \pmod{2}$ e, portanto, todas as raízes quadradas de 1, módulo 2, são triviais. Assim, n não pode ser primo. Finalmente, devemos ter $n > 1$ para que exista uma raiz quadrada não trivial de 1. Então, n tem de ser composto.

Exponenciação com elevação ao quadrado repetida

Uma operação que ocorre frequentemente em cálculos da teoria dos números é a elevação de um número a uma potência módulo outro número, também conhecida como ***exponenciação modular.*** Mais precisamente, o que queremos é um modo eficiente para calcular $a_b \bmod n$, onde a e b são inteiros não negativos e n é um inteiro positivo. A exponenciação modular é uma operação essencial em muitas rotinas de teste de primalidade e no sistema de criptografia de chave pública RSA. O método de ***elevação ao quadrado repetida*** resolve esse problema eficientemente utilizando a representação binária de b.

Seja $\langle b_k, b_k$ - 1, ..., $b_1, b_0\rangle$ a representação binária de b (isto é, a representação binária tem $k + 1$ bits de comprimento, b_k é o bit mais significativo e b_0 é o bit menos significativo). O procedimento a seguir calcula $a_c \bmod n$ à medida que c é aumentada por duplicações e incrementações de 0 a b.

```
MODULAR-EXPONENTIATION(a, b, n)
1  c = 0
2  d = 1
3  seja ⟨b_k, b_{k-1}, ..., b_1, b_0⟩ a representação binária de b
4  for i = k downto 0
5      c = 2c
6      d = (d · d) mod n
7      if b_i == 1
8          c = c + 1
9          d = (d · a) mod n
10 return d
```

O uso essencial da elevação ao quadrado na linha 6 de cada iteração explica o nome "elevação ao quadrado repetida". Como exemplo, para $a = 7$, $b = 560$ e $n = 561$, o algoritmo calcula a sequência de valores módulo 561 mostrada na Figura 31.4; a sequência de expoentes utilizados é mostrada na linha da tabela identificada por c.

Na verdade, o algoritmo não precisa da variável c, mas ela é incluída para o seguinte invariante de laço de duas partes:

Imediatamente antes de cada iteração do laço **for** das linhas 4–9,

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Figura 31.4 Resultados de MODULAR-EXPONENTIATION quando o procedimento calcula a^b (mod n), onde $a = 7$, $b = 560 = \langle 1000110000 \rangle$ e $n = 561$. Os valores são mostrados após cada execução do laço **for**. O resultado final é 1.

1. O valor de c é o mesmo que o do prefixo $\langle b_k, b_{k-1}, ..., b_{i+1} \rangle$ da representação binária de b.
2. $d = a^c \bmod n$.

Usamos esse invariante de laço da seguinte maneira:

Inicialização: Inicialmente, $i = k$, de modo que o prefixo $\langle b_k, b_{k-1}, ..., b_{i+1} \rangle$ está vazio, o que corresponde a $c = 0$. Além disso, $d = 1 = a^0 \bmod n$.

Manutenção: Sejam c' e d' os valores de c e d no final de uma iteração do laço **for** e, portanto, os valores antes da próxima iteração. Cada iteração atualiza $c' = 2c$ (se $b_i = 0$) ou $c' = 2c + 1$ (se $b_i = 1$), de modo que c estará correta antes da próxima iteração. Se $b_i = 0$, então $d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{2c} \bmod n = \bmod n = a^{c'} \bmod n$. Se $b_i = 1$, então $d' = d^2 a \bmod n = (a^c)^2 a \bmod n = a^{2c+1} \bmod n = a^{c'} \bmod n$. Em qualquer caso, $d = a^c \bmod n$ antes da próxima iteração.

Término: No término, $i = -1$. Assim, $c = b$, já que c tem o valor do prefixo $\langle b_k, b_{k-1}, ..., b_0 \rangle$ da representação binária de b. Consequentemente, $d = a^c \bmod n = a^b \bmod n$.

Se as entradas a, b e n são números de bits, então o número total de operações aritméticas exigidas é $O()$, e o número total de operações com bits exigidas é $O(^3)$.

Exercícios

31.6-1 Desenhe uma tabela mostrando a ordem de todos os elementos em $_{11}^*$. Escolha a menor raiz primitiva g e calcule uma tabela que dê $\text{ind}_{11,g}(x)$ para todo $x \in _{11}^*$.

31.6-2 Dê um algoritmo de exponenciação modular que examine os bits de b da direita para a esquerda, em vez de da esquerda para a direita.

31.6-3 Supondo que você conhece $f(n)$, explique como calcular $a^{-1} \bmod n$ para qualquer $a \in _n^*$ usando o procedimento MODULAR-EXPONENTIATION.

31.7 O SISTEMA DE CRIPTOGRAFIA DE CHAVE PÚBLICA RSA

Com um sistema de criptografia de chave pública podemos criptografar mensagens enviadas entre dois participantes de uma comunicação, de modo que um intruso que escute as mensagens criptografadas não possa decifrá-las. Um sistema de criptografia de chave pública também permite que um dos participantes acrescente uma "assinatura digital" impossível de forjar ao final de uma mensagem eletrônica. Tal assinatura é a versão eletrônica de uma assinatura manuscrita em um documento em papel e pode ser facilmente verificada por qualquer pessoa, não pode ser forjada por ninguém e perde sua validade se qualquer bit da mensagem for alterado. Portanto, permite a autenticação da identidade do signatário, bem como do conteúdo da mensagem assinada. É a ferramenta perfeita para assinar eletronicamente

contratos de negócios, cheques eletrônicos, pedidos de compras eletrônicos e outras comunicações eletrônicas que as partes desejem autenticar.

O sistema de criptografia de chave pública RSA se baseia na espetacular diferença entre a facilidade de encontrar números primos grandes e a dificuldade de fatorar o produto de dois números primos grandes. A Seção 31.8 descreve um procedimento eficiente para encontrar números primos grandes, e a Seção 31.9 discute o problema de fatorar inteiros grandes.

Sistemas de criptografia de chave pública

Em um sistema de criptografia de chave pública, cada participante tem uma ***chave pública*** e uma ***chave secreta***. Cada chave é uma informação. Por exemplo, no sistema de criptografia RSA, cada chave consiste em um par de inteiros. Os participantes "Alice" e "Bob" são tradicionalmente usados em exemplos de criptografia; denotamos suas chaves públicas e secretas como P_A, S_A para Alice e P_B, S_B para Bob.

Cada participante cria suas próprias chaves pública e secreta. Chaves secretas são mantidas em segredo, mas chaves públicas podem ser reveladas a qualquer um ou até divulgadas publicamente. Na verdade, muitas vezes, é conveniente supor que a chave pública de qualquer pessoa está disponível em um diretório público, de modo que qualquer participante possa obter facilmente a chave pública de qualquer outro participante.

As chaves pública e secreta especificam funções que podem ser aplicadas a qualquer mensagem. Seja D o conjunto de mensagens permissíveis. Por exemplo, D poderia ser o conjunto de todas as sequências de bits de comprimento finito. Na formulação mais simples e original da criptografia de chave pública, as chaves pública e secreta devem especificar funções biunívocas de D para ele próprio. Denotamos a função correspondente à chave pública de Alice, P_A, por $P_A(\ )$ e a função correspondente à sua chave secreta, S_A, por $S_A(\ )$. Portanto, as funções $P_A(\ )$ e $S_A(\ )$ são permutações de D. Supomos que as funções $P_A(\ )$ e $S_A(\ )$ podem ser calculadas eficientemente, dada a chave correspondente P_A ou S_A.

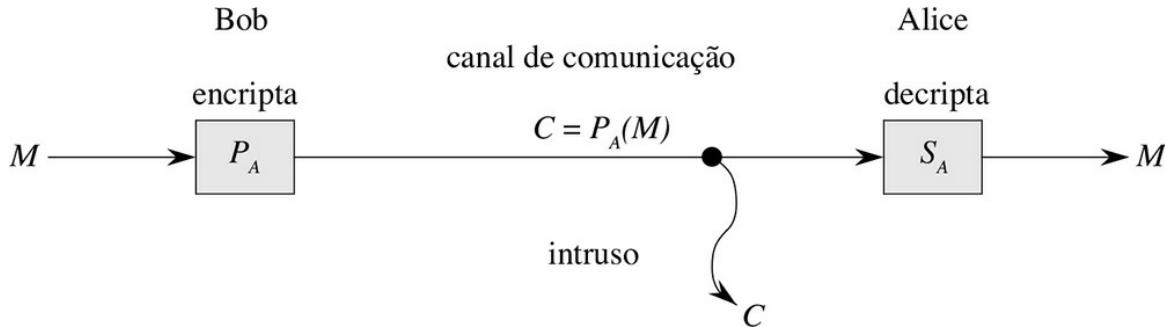
As chaves pública e secreta para qualquer participante formam um "par compatível", já que especificam funções que são inversas uma da outra. Isto é,

$$M = S_A(P_A(M)) \ , \tag{31.35}$$

$$M = P_A(S_A(M)) \tag{31.36}$$

para qualquer mensagem $M \in$ D. Transformar M com as duas chaves P_A e S_A, sucessivamente, em qualquer ordem, produz novamente a mensagem M.

Em um sistema de criptografia de chave pública, é essencial que ninguém, exceto Alice, possa calcular a função $S_A(\ )$ em qualquer período de tempo prático. Tal proposição é crucial para manter a privacidade das mensagens criptografadas enviadas a Alice e para saber que as assinaturas digitais de Alice são autênticas. Alice deve manter S_A em segredo; se ela não o fizer, perderá a exclusividade da chave, e o sistema de criptografia não poderá lhe oferecer recursos exclusivos. A pressuposição de que somente Alice pode calcular $S_A(\ )$ deve se manter válida mesmo que todos conheçam P_A e possam calcular eficientemente $P_A(\ )$, a função inversa de $S_A(\ )$. Para projetar um sistema de criptografia de chave pública funcional, temos de formular um sistema no qual possamos revelar uma transformação $P_A(\ )$, sem com isso revelar como calcular a transformação inversa $S_A(\ )$ correspondente. Essa tarefa parece descomunal, mas veremos como podemos executá-la.

Em um sistema de criptografia de chave pública, a codificação funciona como mostra a Figura 31.5. Suponha que Bob queira enviar a Alice uma mensagem M criptografada de tal forma que ela pareça algaravia ininteligível para um intruso. O cenário para enviar a mensagem é dado a seguir.

- Bob obtém a chave pública de Alice, P_A (de um diretório público ou diretamente de Alice).
- Bob calcula o ***texto cifrado*** $C = P_A(M)$ correspondente à mensagem M e envia C a Alice.

- Quando recebe o texto cifrado C, Alice aplica sua chave secreta S_A para recuperar a mensagem original: $S_A(C) = S_A(P_A(M)) = M$.

Como $S_A(\,)$ e $P_A(\,)$ são funções inversas, Alice pode calcular M a partir de C. Visto que somente Alice pode calcular $S_A(\,)$, ela é a única pessoa que pode calcular M a partir de C. Como Bob criptografa M usando $P_A(\,)$, só Alice pode entender a mensagem transmitida.

Podemos implementar assinaturas digitais com igual facilidade dentro da nossa formulação de um sistema de criptografia de chave pública. (Há outros modos de abordar o problema de construir assinaturas digitais, mas não os examinaremos aqui.) Suponha agora que Alice queira enviar a Bob uma resposta M' com assinatura digital. A Figura 31.6 mostra como ocorre a assinatura digital.

- Alice calcula sua ***assinatura digital*** s para a mensagem M' usando sua chave secreta S_A e a equação $s = S_A(M')$.

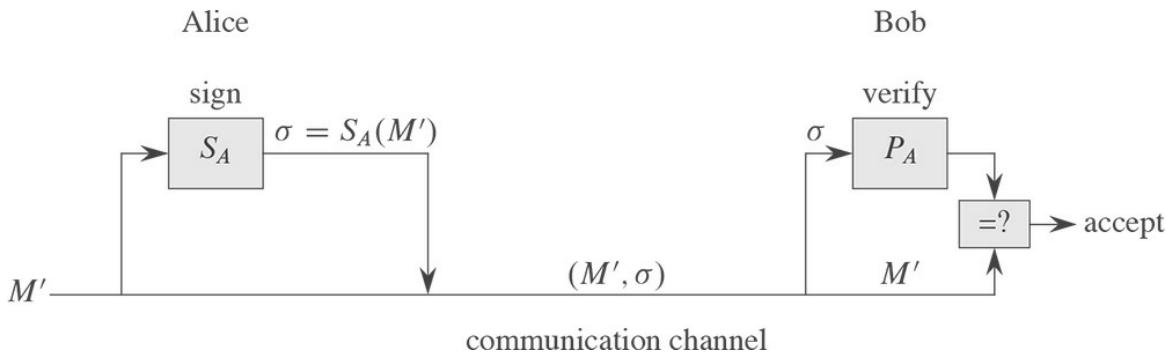



Figura 31.5 Codificação em um sistema de chave pública. Bob codifica a mensagem M usando a chave pública P_A de Alice e transmite o texto cifrado resultante $C = P_A(M)$ a Alice por um canal de comunicação. Um intruso que capturar o texto cifrado transmitido não obterá nenhuma informação sobre M. Alice recebe C e o decifra usando sua chave secreta para obter a mensagem original $M = S_A(C)$.

- Alice envia o par mensagem/assinatura (M', s) a Bob.
- Ao receber (M', s), Bob pode confirmar que ela foi enviada por Alice utilizando a chave pública de Alice para verificar a equação $M' = P_A(s)$. (Presumivelmente, M' contém o nome de Alice e, assim, Bob sabe qual chave pública deve usar.) Se a equação é válida, Bob conclui que a mensagem M' foi realmente assinada por Alice. Se a equação não é válida, Bob conclui que a mensagem M' ou a assinatura digital s foi danificada por erros de transmissão ou que o par (M', s) é uma tentativa de falsificação.

Como proporciona autenticação da identidade do signatário, bem como autenticação do conteúdo da mensagem assinada, uma assinatura digital é análoga a uma assinatura manuscrita no final de um documento escrito.

Uma assinatura digital tem de ser verificável por quem quer que tenha acesso à chave pública do signatário. Uma mensagem assinada pode ser verificada por uma parte e depois repassada a outras partes que também podem verificar a assinatura. Por exemplo, a mensagem poderia ser um cheque eletrônico de Alice para Bob. Depois de verificar a assinatura de Alice no cheque, Bob pode entregá-lo a seu banco, que então também pode verificar a assinatura e efetuar a transferência de fundos adequada.

Uma mensagem assinada não está necessariamente criptografada; a mensagem pode estar "às claras" e não protegida contra revelação. Compondo os protocolos já citados para codificação e para assinaturas, podemos criar mensagens assinadas e criptografadas. Primeiro, o signatário anexa sua assinatura digital à mensagem e depois criptografa o par mensagem/assinatura resultante com a chave pública do destinatário pretendido. O destinatário decodifica a mensagem recebida com sua chave secreta para obter a mensagem original e também sua assinatura digital. Então pode verificar a assinatura usando a chave pública do signatário. O processo combinado correspondente quando são utilizados sistemas em papel é assinar o documento em papel e depois lacrá-lo dentro de um envelope de papel que será aberto apenas pelo destinatário pretendido.

Figura 31.6 Assinaturas digitais em um sistema de chave pública. Alice assina a mensagem M' anexando a ela sua assinatura digital $s = S_A(M')$. Ela transmite o par mensagem/assinatura (M', s) a Bob, que a verifica conferindo a equação $M' = P_A(s)$. Se a equação é válida, ele aceita (M', s) como uma mensagem assinada por Alice.

O sistema de criptografia RSA

No **sistema de criptografia de chave pública RSA**, um participante cria suas chaves pública e secreta com o seguinte procedimento:

1. Selecione aleatoriamente dois números primos grandes p e q, tais que p q. Os primos p e q podem ter, digamos, 512 bits cada um.
2. Calcule $n = pq$.
3. Selecione um inteiro ímpar pequeno e tal que e seja primo com $f(n)$, que, pela equação (31.20), é igual a $(p - 1)(q - 1)$.
4. Calcule d como o inverso multiplicativo de e, módulo $f(n)$. (O Corolário 31.26 garante que d existe e é definido unicamente. Podemos usar a técnica da Seção 31.4 para calcular d, dados e e $f(n)$.)
5. Divulgue o par $P = (e, n)$ como a ***chave pública RSA*** do participante.
6. Mantenha o par $S = (d, n)$ em segredo como a ***chave secreta RSA*** do participante.

Por esse esquema, o domínio D é o conjunto $_n$. Para transformar uma mensagem M associada a uma chave pública $P = (e, n)$ calcule

$$P(M) = M^e \pmod{n} . \qquad (31.37)$$

Para transformar um texto cifrado C associado a uma chave secreta $S = (d, n)$, calcule

$$S(C) = C^d \bmod n. \qquad (31.38)$$

Essas equações se aplicam à codificação, bem como a assinaturas. Para criar uma assinatura, o signatário aplica sua chave secreta à mensagem a ser assinada, em vez de a um texto cifrado. Para verificar uma assinatura, a chave pública do signatário é aplicada a ela, em vez de ser aplicada a uma mensagem a ser criptografada.

Podemos implementar as operações de chave pública e chave secreta usando o procedimento MODULAR-EXPONENTIATION descrito na Seção 31.6. Para analisar o tempo de execução dessas operações, suponha que a chave pública (e, n) e a chave secreta (d, n) satisfaçam $\lg e = O(1)$, $\lg d \leq$ e $\lg n \leq$. Então, aplicar uma chave pública requer $O(1)$ multiplicações modulares e usa $O(^2)$ operações com bits. Aplicar uma chave secreta requer $O()$ multiplicações modulares, usando $O(^3)$ operações com bits.

Teorema 31.36 (Correção do RSA)

As equações RSA (31.37) e (31.38) definem transformações inversas de $_n$ que satisfazem as equações (31.35) e (31.36).

Prova Pelas equações (31.37) e (31.38) temos que, para qualquer $M \in$ $_n$

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n} .$$

Visto que e e d são inversos multiplicativos módulo $f(n) = (p - 1)(q - 1)$,

$$ed = 1 + k(p-1)(q-1)$$

para algum inteiro k. Mas, então, se M 0 (mod p), temos

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} && \pmod{p} \\ &\equiv M(M \bmod p)^{p-1})^{k(q-1)} && \pmod{p} \quad \text{(pelo Teorema 31.31)} \\ &\equiv M(1)^{k(q-1)} && \pmod{p} \\ &\equiv M && \pmod{p} \end{aligned}$$

Além disso, $M^{ed} \equiv M \pmod{p}$ se $M \equiv 0 \pmod{p}$. Assim,
$M^{ed} \equiv M \pmod{p}$
para todo M. De modo semelhante,
$M^{ed} \equiv M \pmod{q}$
para todo M. Portanto, pelo Corolário 31.29 do teorema chinês do resto,
$M^{ed} \equiv M \pmod{n}$
para todo M.

A segurança do sistema de criptografia RSA se baseia em grande parte na dificuldade de fatorar inteiros grandes. Se um adversário pode fatorar o módulo n em uma chave pública, então pode deduzir a chave secreta a partir da chave pública usando o conhecimento dos fatores p e q do mesmo modo que o criador da chave pública os usou. Portanto, se for fácil fatorar inteiros grandes, então é fácil quebrar o sistema de criptografia RSA. A afirmação inversa, isto é, se for difícil fatorar inteiros grandes é difícil quebrar o RSA, não foi provada. Todavia, depois de duas décadas de pesquisas, não foi encontrado nenhum método mais fácil para quebrar o sistema de criptografia de chave pública RSA do que fatorar o módulo n. E, como veremos na Seção 31.9, fatorar inteiros grandes é surpreendentemente difícil. Selecionando aleatoriamente e multiplicando dois primos de 1024 bits, podemos criar uma chave pública que não poderá ser "quebrada" em nenhum período de tempo viável com a tecnologia atual. Na ausência de inovação fundamental no projeto de algoritmos da teoria dos números, e quando implementado com cuidado de acordo com padrões recomendados, o sistema de criptografia RSA pode garantir alto grau de segurança em aplicações.

Contudo, para conseguir segurança com o sistema de criptografia RSA, devemos trabalhar com inteiros muito longos — centenas de bits de comprimento ou até mais, para resistir a possíveis avanços na arte da fatoração. À época da redação deste livro (2009), os módulos RSA encontravam-se comumente na faixa de 768 a 2048 bits. Para criar módulos desses tamanhos, temos de determinar primos grandes eficientemente. A Seção 31.8 aborda esse problema.

Por questão de eficiência, muitas vezes, o RSA é usado de um modo "híbrido" ou de "gerenciamento de chaves", com sistemas de criptografia rápidos de chaves não públicas. Com um sistema desse tipo, as chaves de codificação e decodificação são idênticas. Se Alice desejar enviar a Bob uma longa mensagem secreta M, ela seleciona uma chave aleatória K para o sistema de criptografia rápido de chave não pública e criptografa M usando K, obtendo o texto cifrado C. Nesse caso, C é tão longo quanto M, mas K é bem curto. Em seguida, ela codifica K usando a chave pública RSA de Bob. Visto que K é curta, calcular $P_B(K)$ é rápido (muito mais rápido que calcular $P_B(M)$). Então, ela transmite $(C, P_B(K))$ a Bob, que decodifica $P_B(K)$ para obter K e depois usa K para decodificar C, obtendo M.

Podemos usar uma abordagem híbrida semelhante para criar assinaturas digitais eficientemente. Essa abordagem combina RSA com uma ***função hash resistente a colisões*** h uma função fácil de calcular, mas para a qual é inviável, em termos computacionais, encontrar duas mensagens M e M' tais que $h(M) = h(M')$. O valor $h(M)$ é uma "impressão digital" curta (digamos, de 256 bits) da mensagem M. Se Alice deseja assinar uma mensagem M, em

primeiro lugar aplicará h a M para obter a impressão digital $h(M)$ que, então, codificará com sua chave secreta. Ela envia $(M, S_A(h(M)))$ a Bob como sua versão assinada de M. Bob pode confirmar a assinatura calculando $h(M)$ e verificando que P_A aplicada a $S_A(h(M))$ como foi recebida é igual a $h(M)$. Como ninguém poder criar duas mensagens com a mesma impressão digital, é impossível, em termos computacionais, alterar uma mensagem assinada e ainda assim preservar a validade da assinatura.

Finalmente, observamos que o uso de ***certificados*** torna muito mais fácil a distribuição de chaves públicas. Por exemplo, suponha que existe uma "autoridade confiável" T cuja chave pública é conhecida por todos. Alice pode obter de T uma mensagem assinada (seu certificado) que declara que "a chave pública de Alice é P_A". Esse certificado é "autoautenticado", já que todos conhecem P_T. Alice pode incluir seu certificado em suas mensagens assinadas, de modo que sua chave pública fique imediatamente disponível para o destinatário, que assim pode verificar a assinatura da remetente. Visto que a chave pública de Alice foi assinada por T, o destinatário sabe que a chave de Alice é realmente de Alice.

Exercícios

31.7-1 Considere um conjunto de chaves RSA com $p = 11$, $q = 29$, $n = 319$ e $e = 3$. Que valor de d deve ser usado na chave secreta? Qual é a codificação da mensagem $M = 100$?

31.7-2 Prove que, se o expoente público e de Alice é 3 e um adversário obtém o expoente secreto d de Alice, onde $0 < d < f(n)$, então o adversário pode fatorar o módulo n de Alice em tempo polinomial em relação ao número de bits em n. (Se bem que não pedimos que você o prove, seria interessante saber que esse resultado continua verdadeiro mesmo que a condição $e = 3$ seja eliminada.) Consulte Miller [255].)

31.7-3 ★ Prove que o RSA é multiplicativo, no sentido de que

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n} .$$

Use esse fato para provar que, se um adversário tivesse um procedimento que conseguisse decifrar eficientemente 1% das mensagens em $_n$ criptografadas com P_A, ele poderia empregar um algoritmo probabilístico para decifrar todas as mensagens criptografadas com P_A com alta probabilidade.

31.8 ★ TESTE DE PRIMALIDADE

Nesta seção, consideraremos o problema de encontrar primos grandes. Começamos com uma discussão da densidade de primos, em seguida, examinamos uma abordagem plausível (embora incompleta) para testar primalidade e depois apresentamos um teste de primalidade aleatorizado efetivo criado por Miller e Rabin.

Densidade de números primos

Para muitas aplicações, como a criptografia, precisamos encontrar primos grandes "aleatórios". Felizmente, primos grandes não são muito raros, de modo que é viável testar inteiros aleatórios do tamanho adequado até encontrar um primo. A ***função distribuição de primos*** (n) especifica o número de primos menores ou iguais a n. Por exemplo, (10) = 4, já que existem quatro números primos menores ou iguais a 10, isto é, 2, 3, 5 e 7. O teorema dos números primos nos dá uma aproximação útil para (n).

Teorema 31.37 (Teorema dos números primos)

$$\lim_{n \to \infty} \frac{\pi(n)}{n / \ln n} = 1.$$

A aproximação $n/\ln n$ produz estimativas razoavelmente precisas de (n), mesmo para n pequeno. Por exemplo, ela erra por menos de 6% em $n = 10^9$, onde (n) = 50.847.534 e $n/\ln n$ 48.254.942. (Para um especialista em teoria dos números, 10^9 é um número pequeno.)

Podemos ver o processo de selecionar aleatoriamente um inteiro n e determinar se ele é primo como uma tentativa de Bernoulli (veja Seção C.4). Pelo teorema dos números primos, a probabilidade de sucesso — isto é, a probabilidade de n ser primo — é aproximadamente $1/ \ln n$. A distribuição geométrica nos diz quantas tentativas precisamos para obter um sucesso e, pela equação (C.32), o número esperado de tentativas é aproximadamente $\ln n$. Assim, esperaríamos examinar aproximadamente $\ln n$ inteiros escolhidos aleatoriamente próximos de n para encontrar um primo com o mesmo comprimento de n. Por exemplo, esperamos que encontrar um primo de 1.024 bits exigiria testar a primalidade de aproximadamente $\ln 2^{1024} \approx 710$ números de 1.024 bits escolhidos aleatoriamente (claro que podemos reduzir esse número à metade escolhendo somente inteiros ímpares).

No restante desta seção, consideraremos o problema de determinar se um inteiro ímpar grande n é ou não primo. Por conveniência de notação, supomos que a fatoração de n em primos é

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} \qquad (31.39)$$

onde $r \geq 1$, $p_1, p_2, ..., p_r$ são os fatores primos de n, e $e_1, e_2, ..., e_r$ são inteiros positivos. O inteiro n é primo se e somente se $r = 1$ e $e_1 = 1$.

Uma abordagem simples para o problema de testar a primalidade é a ***divisão experimental***. Tentamos dividir n por cada inteiro 2, 3, ... $\sqrt{n}$ (novamente, podemos ignorar inteiros pares maiores que 2). É fácil ver que n é primo se e somente se nenhum dos divisores experimentais divide n. Considerando que cada divisão experimental demora um tempo constante, o tempo de execução do pior caso é $\Theta(\sqrt{n})$, que é exponencial em relação ao comprimento de n. (Lembre-se de que, se n for codificado em binário usando bits, então = lg (n + 1), e portanto $\sqrt{n} = \Theta(2^{/2})$.) Assim, a divisão experimental funciona bem somente se n é muito pequeno ou se por acaso ele tem um fator primo pequeno. Quando funciona, a divisão experimental tem a vantagem de não somente determinar se n é primo ou composto, mas também de determinar um dos fatores primos de n, se n for composto.

Nesta seção, estamos interessados apenas em determinar se um dado número n é primo; se n for composto, não nos preocuparemos em determinar seus fatores primos. Como veremos na Seção 31.9, calcular os fatores primos de um número é dispendioso em termos computacionais. Talvez seja surpreendente que seja muito mais fácil saber se um dado número é primo ou não que determinar os fatores primos do número se ele não for primo.

Teste de pseudoprimalidade

Agora, consideramos um método para testar primalidade que "quase funciona" e que, na verdade, é bom o bastante para muitas aplicações práticas. Mais adiante apresentaremos um refinamento desse método que elimina o pequeno defeito. Representamos por ^+_n os elementos não nulos de $_n$:

$$\mathbb{Z}_n^+ = \{1, 2, ..., n - 1\} .$$

Se n é primo, então $^+_n = ^*_n$.

Dizemos que n é um ***pseudoprimo de base a*** se n é composto e

$$a^{n-1} \equiv 1 \pmod{n} . \qquad (31.40)$$

O teorema de Fermat (Teorema 31.31) implica que, se n é primo, então n satisfaz a equação (31.40) para todo a em $+_n$. Assim, se pudermos encontrar qualquer $a \in +_n$ tal que n *não* satisfaz a equação (31.40), então n é certamente composto. É surpreendente que a recíproca *quase* seja válida, de modo que esse critério forma um teste quase perfeito de primalidade. Testamos para verificar se n satisfaz a equação (31.40) para $a = 2$. Se não, declaramos que n *é* composto retornando COMPOSITE. Caso contrário, retornamos PRIME, adivinhando que n é primo (quando, na verdade, tudo o que sabemos é que n é primo ou é um pseudoprimo de base 2).

O procedimento a seguir pretende verificar a primalidade de n desse modo. Utiliza o procedimento MODULAR-EXPONENTIATION da Seção 31.6. A entrada n supõe que n seja um inteiro ímpar maior que 2.

```
PSEUDOPRIME(n)
1  if MODULAR-EXPONENTIATION(2, n − 1, n) ≢1 (mod n)
2       return COMPOSITE               // definitivamente
3  else return PRIME                   // esperamos!
```

Esse procedimento pode cometer erros, mas somente de um tipo. Isto é, se o procedimento afirmar que n é composto, então está sempre correto. Porém, se afirmar que n é primo, comete um erro somente se n é um pseudoprimo base 2.

Qual é a frequência de erro desse procedimento? Surpreendentemente rara. Há somente 22 valores de n menores que 10.000 para os quais ele erra; os quatro primeiros desses valores são 341, 561, 645 e 1105. Não provaremos, mas a probabilidade de esse programa cometer um erro em um número de bits escolhido aleatoriamente tende a zero quando $\rightarrow \infty$. Utilizando estimativas mais precisas criadas por Pomerance [279] do número de pseudoprimos de base 2 de um dado tamanho, podemos avaliar que a probabilidade de um número de 512 bits escolhido aleatoriamente e considerado primo pelo procedimento dado ser um pseudoprimo de base 2 é menor do que uma vez em 10^{20} e a probabilidade de um número de 1.024 bits escolhido aleatoriamente e considerado primo ser um pseudoprimo de base 2 é menor que um em 10^{41}. Portanto, se estiver simplesmente tentando encontrar um número primo grande para alguma aplicação, para todos os propósitos práticos você quase nunca estará errado se escolher números grandes aleatoriamente até que, para um deles, PSEUDOPRIME retorne PRIME. Porém, quando os números cuja primalidade está sendo testada não são escolhidos aleatoriamente, precisamos de uma abordagem melhor para testar a primalidade. Como veremos, um pouco mais de esperteza e alguma aleatoriedade produzirão uma rotina de teste de primalidade que funciona bem para todas as entradas.

Infelizmente, não podemos eliminar completamente todos os erros simplesmente verificando a equação (31.40) para um segundo número-base, digamos $a = 3$, porque existem inteiros compostos n, conhecidos como ***números de Carmichael***, que satisfazem a equação (31.40) para *todo* $a \in {}^*_n$. (Observamos que a equação (31.40) falha quando mdc $(a, n) > 1$ — isto é, quando $a \in {}^*_n$ —, mas esperar demonstrar que n é composto determinando tal a pode ser difícil se n tiver somente fatores primos grandes.) Os três primeiros números de Carmichael são 561, 1105 e 1729. Números de Carmichael são extremamente raros; há, por exemplo, apenas 255 deles menores que 100.000.000. O Exercício 31.8-2 ajuda a explicar por que eles são tão raros.

Em seguida, mostramos como melhorar nosso teste de primalidade de modo que ele não seja enganado por números de Carmichael.

Teste aleatório de primalidade de Miller-Rabin

O teste de primalidade de Miller-Rabin supera os problemas do teste simples PSEUDOPRIME com duas modificações:

- Experimenta diversos valores-base a escolhidos aleatoriamente, em vez de apenas um valor-base.
- Enquanto calcula cada exponenciação modular, procura uma raiz quadrada não trivial de 1, módulo n, durante o teste final das elevações ao quadrado. Se encontrar alguma, para e retorna COMPOSITE. O Corolário 31.35 na Seção 31.6 justifica a detecção de compostos feita dessa maneira.

O pseudocódigo para o teste de primalidade de Miller-Rabin é dado a seguir. A entrada $n > 2$ é o número ímpar cuja primalidade será testada, e s é o número de valores-base escolhidos aleatoriamente de $+_n$ que serão experimentados. O código utiliza o gerador de número aleatórios RANDOM descrito na página 117: RANDOM(1, $n - 1$) retorna um inteiro a escolhido aleatoriamente que satisfaz $1 \leq a \leq n - 1$. O código emprega um procedimento auxiliar WITNESS tal que WITNESS(a, n) é TRUE SE e somente se a é uma "testemunha" de que n é realmente composto isto é, se é possível usar a para provar (de um modo que veremos em breve) que n é composto. O teste WITNESS(a, n) é uma extensão, porém mais eficiente, do teste

$$a^{n-1} \not\equiv 1 \pmod{n}$$

que formou a base (usando $a = 2$) para PSEUDOPRIME. Primeiro, apresentamos e justificamos a construção de SC, e depois mostramos como o utilizamos no teste de primalidade de Miller-Rabin. Seja $n - 1 = 2^{tu}$, onde $t \geq 1$ e u é ímpar; isto é, a representação binária de $n - 1$ é a representação binária do inteiro ímpar u seguido por exatamente t zeros. Portanto, $a_n{}^{-1} \equiv (a_u)^{2t}$, de modo que podemos calcular a_n - ^{1} mod n calculando primeiro a_u mod n e depois elevando o resultado ao quadrado t vezes sucessivamente.

```
WITNESS(a,n)
1  sejam t e u tais que t ≥ 1, u é ímpar e n − 1 = 2^t u
2  x_0 = MODULAR-EXPONENTIATION(a, u, n)
3  for i = 1 to t
4       x_i = x_{i−1}^2 mod n
5       if x_i == 1 e x_{i − 1} ≠ 1 e x_{i − 1} ≠ n − 1
6            return TRUE
7  if x_t ≠ 1
8       return TRUE
9  return FALSE
```

Esse pseudocódigo para WITNESS calcula a_n - ^{1} mod n calculando primeiro o valor $x = a_u$ mod n na linha 2 e depois elevando o resultado ao quadrado t vezes em sequência no laço **for** das linhas 3–6. Por indução em relação a i, a sequência $x_0, x_1, ..., x_t$ de valores calculados satisfaz a equação $x_i \equiv a_2{}^{iu}$ (mod n) para $i = 0, 1, ..., t$, de modo que, em particular, $x_t \equiv a_n$ - ^{1} (mod n). Contudo, depois que a linha 4 executa uma etapa de elevação ao quadrado, o laço pode terminar prematuramente se as linhas 5–6 perceberem que uma raiz quadrada não trivial de 1 foi encontrada. (Mais adiante explicaremos esse teste.) Se isso ocorrer, o algoritmo para e devolve TRUE. As linhas 7–8 retornam TRUE se o valor calculado para $x_t \equiv a_n$ - ^{1} (mod n) não é igual a 1, exatamente como o procedimento PSEUDOPRIME devolve COMPOSITE nesse caso. A linha 9 devolve FALSE SE não tivermos devolvido TRUE na linha 6 ou 8.

Agora, demonstramos que, se WITNESS(a, n) devolve TRUE, podemos construir uma prova de que n é composto usando a como testemunha.

Se WITNESS devolve TRUE da linha 8, então descobriu que $x_t = a_n$ - ^{1} mod $n \neq 1$. Contudo, se n é primo, temos, pelo teorema de Fermat (Teorema 31.31), que a_n - ${}^1 \equiv 1$ (mod n) para todo $a \in +_n$ Portanto, n não pode ser primo, e a equação a_n - ^{1} mod $n \neq 1$ prova esse fato.

Se WITNESS retorna TRUE da linha 6, então descobriu que x_i - ^{1} é uma raiz quadrada não trivial de 1, módulo n, visto que temos que x_i - ^{1} ± 1 (mod n) apesar de $x_i \equiv x_2{}^{i}$- ${}^1 \equiv 1$ (mod n). O Corolário 31.35 afirma que somente se n é composto pode existir uma raiz quadrada não trivial de 1 módulo n, de modo que demonstrar que x_i - ^{1} é uma raiz quadrada não trivial de 1 módulo n prova que n é composto.

Isso conclui nossa prova da correção de WITNESS. Se verificarmos que a chamada WITNESS(a, n) devolve TRUE, então n certamente é composto, e a testemunha a, aliada a devolução de TRUE pelo procedimento (ele retornou da linha 6 ou da linha 8?) prova que n é composto.

Nesse ponto, apresentamos uma descrição alternativa resumida do comportamento de WITNESS em função da sequência $X = \langle x_0, x_1, ..., x_t \rangle$, que mais tarde verificaremos ser útil quando analisarmos a eficiência do teste de primalidade de Miller-Rabin. Observe que, se $x_i = 1$ para algum $0 \leq i < t$, WITNESS poderia não calcular o restante da sequência. Porém, se o fizesse, cada valor $x_i + 1$, $x_i + 2$, ..., x_t seria 1, e supomos que essas posições na sequência X são todas iguais a 1. Temos quatro casos:

1. $X = \langle ..., d \rangle$, onde d 1: a sequência X não termina em 1. Devolva TRUE na linha 8; a é uma testemunha de que n é composto (pelo Teorema de Fermat).
2. $X = \langle 1, 1, ..., 1 \rangle$: a sequência X é toda formada por 1. Devolva FALSE; a não é uma testemunha de que n é composto.
3. $X = \langle ..., -1, 1, ..., 1 \rangle$: a sequência X termina em 1, e o último valor não 1 é igual a -1. Devolva FALSE; a não é uma testemunha de que n é composto.
4. $X = \langle ..., d, 1, ..., 1 \rangle$, onde d ±1: a sequência X termina em 1, mas o último valor não 1 não é –1. Devolva TRUE na linha 6; a é uma testemunha de que n é composto, já que d é uma raiz quadrada não trivial de 1.

Agora, examinamos o teste de primalidade de Miller-Rabin com base no uso de WITNESS. Novamente, supomos que n seja um inteiro ímpar maior que 2.

```
MILLER-RABIN(n, s)
1  for j = 1 to s
2      a = RANDOM(1, n − 1)
3      if WITNESS(a, n)
4          return COMPOSITE            // definitivamente
5  return PRIME                        // quase certamente
```

O procedimento MILLER-RABIN é uma busca probabilística de uma prova de que n é composto. O laço principal (que começa na linha 1) escolhe até s valores aleatórios de a de + (linha 2). Se um dos valores de a escolhidos for uma testemunha de que n é composto, então MILLER-RABIN devolve COMPOSITE na linha 4. Tal resultado é sempre correto, pela correção de WITNESS. Se MILLER-RABIN não encontrar nenhuma testemunha em s tentativas, entenderá que isso ocorreu porque não existe nenhuma testemunha e, portanto, deduz que n é primo. Veremos que esse resultado é provavelmente correto se s for suficientemente grande, mas que ainda há uma minúscula chance de o procedimento ter sido infeliz em sua escolha dos valores de a e que as testemunhas realmente existem, ainda que nenhuma tenha sido encontrada.

Para ilustrar a operação de MILLER-RABIN, seja n o número de Carmichael 561, de modo que $n - 1 = 560 = 24 \cdot 35$, $t = 4$ e $u = 35$. Se o procedimento escolher $a = 7$ como base, a Figura 31.4 na Seção 31.6 mostra que WITNESS calcula $x_0 \equiv a^{35} \equiv 241 \pmod{561}$ e, portanto, calcula a sequência $X = \langle 241, 298, 166, 67, 1 \rangle$. Assim, WITNESS descobre uma raiz quadrada não trivial de 1 na última etapa de elevação ao quadrado, já que $a_{280} \equiv 67$ (mod n) e $a_{560} \equiv 1$ (mod n). Então, $a = 7$ é uma testemunha de que n é composto, WITNESS(7, n) retorna TRUE E MILLER-RABIN retorna COMPOSITE.

Se n é um número de bits, MILLER-RABIN requer $O(s)$ operações aritméticas e $O(s^3)$ operações com bits, já que assintoticamente não requer mais trabalho que s exponenciações modulares.

Taxa de erro do teste de primalidade de Miller-Rabin

Se MILLER-RABIN devolve PRIME, há uma chance muito exígua de o procedimento ter cometido um erro. Porém, diferentemente de PSEUDOPRIME, a chance de erro não depende de n; não há nenhuma entrada ruim para esse procedimento. Mais exatamente, ela depende do tamanho de s e da "sorte no jogo" na escolha dos valores-base a. Além disso, visto que cada teste é mais rigoroso que uma simples verificação da equação (31.40), podemos esperar, em princípios gerais, que a taxa de erro seja pequena para inteiros n escolhidos aleatoriamente. O teorema a seguir apresenta um argumento mais preciso.

Teorema 31.38

Se n é um número composto ímpar, o número de testemunhas de que n é composto é pelo menos $(n – 1)/2$.

Prova A prova mostra que o número de não testemunhas é no máximo $(n – 1)/2$, o que implica o teorema.

Começamos afirmando que qualquer não testemunha deve ser um membro de *. Por quê? Considere qualquer não testemunha a. Ela deve satisfazer a_n - 1 ≡ 1 (mod n) ou, o que é equivalente, $a \cdot a_n$ - 2 ≡ 1 (mod n). Portanto, a equação $ax \equiv 1 \pmod{n}$, tem uma solução, a saber, a_n–2. Pelo Corolário 31.21, mdc(a, n) | 1, o que, por sua vez, implica que mdc(a, n) = 1. Portanto, a é um membro de *, todas as não testemunhas pertencem a $*_n$.

Para concluir a prova, mostramos que não apenas todas as não testemunhas estão contidas em $*_n$, mas todas elas estão contidas em um subgrupo próprio B de *. (lembre-se de que dizemos que B é um subgrupo *próprio* de $*_n$ quando B é um subgrupo de $*_n$, mas B não é igual a $*_n$). Então, pelo Corolário 31.16, temos $|B| \leq | *_n |/2$. Visto que $*_n \leq n – 1$, obtemos $|B| \leq (n – 1)/2$. Portanto, o número de não testemunhas é no máximo $(n – 1)/2$ e, assim, o número de testemunhas deve ser, no mínimo, $(n – 1)/2$.

Agora, mostramos como encontrar um subgrupo próprio B de $*_n$, que contém todas as não testemunhas. Dividiremos a prova em dois casos.

Caso 1: Existe um $x \in *_n$ tal que

$$x^{n-1} \not\equiv 1 \pmod{n} .$$

Em outras palavras, n não é um número de Carmichael. Como observamos antes, os números de Carmichael são extremamente raros e, por isso, o Caso 1 é o caso principal que surge "na prática" (por exemplo, quando n foi escolhido aleatoriamente e sua primalidade está sendo testada).

Seja B = {$B \in *_n$: b_n - 1 ≡ 1 (mod n)}. Claramente, B é não vazio, já que $1 \in B$. Visto que B é fechado para multiplicação módulo n, temos que B é um subgrupo de $*_n$. pelo Teorema 31.14. Observe que toda não testemunha pertence a B, já que uma não testemunha a satisfaz a_n - 1 ≡ 1 (mod n). Como $x \in *_n – B$, temos que B é um subgrupo próprio de $*_n$.

Caso 2: Para todo $x \in *_n$,

$$x^{n-1} \equiv 1 \pmod{n} \tag{31.41}$$

Em outras palavras, n é um número de Carmichael. Esse caso é extremamente raro na prática. Contudo, o teste de Miller-Rabin (diferentemente do teste de pseudoprimalidade) pode determinar eficientemente que números de Carmichael são compostos, como mostramos agora.

Nesse caso, n não pode ser uma potência de primo. Para ver por que, vamos supor, por contradição, que $n = p_e$, onde p é um primo e $e > 1$. Deduzimos uma contradição da seguinte maneira: visto que supomos que n é ímpar, p também deve ser ímpar. O Teorema 31.32 implica que $*n$ = é um grupo cíclico: contém um gerador g tal que ord (g) = | $*n$ = |= $f(n) = p_e(1 – 1/p) = (p – 1)\ p_e$ - 1. Pela equação (31.41), temos g_n - 1 ≡ 1 (mod n). Então, o teorema do logaritmo discreto (Teorema 31.33, tomando $y = 0$) implica que $n – 1 \equiv 0$ (mod $f(n)$) ou

$$(p-1)p^{e-1} \mid p^e - 1 .$$

Isso é uma contradição para $e > 1$, já que $(p – 1)p_e$ - 1 é divisível pelo primo p, mas $p_e – 1$ não é. Assim, n não é uma potência de primo.

Visto que o número composto ímpar n não é uma potência de primo, decompomos esse número em um produto $n_1 n_2$, onde n_1 e n_2 são números ímpares maiores que 1 e primos entre si. (Pode haver várias maneiras de decompor n, e não importa qual delas escolhemos. Por exemplo, se $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, então podemos escolher $n_1 = p_1^{e_1}$ and n_2 e $p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r} \cdot$)

Lembre-se de que definimos t e u de modo que $n - 1 = 2^t u$, onde $t \geq 1$ e u é ímpar, e que, para uma entrada a, o procedimento WITNESS calcula a sequência

$$X = \langle a^u, a^{2u}, a^{2^2 u}, ..., a^{2^t u} \rangle$$

(todos os cálculos são executados em módulo n).

Vamos denominar um par de inteiros (v, j) ***aceitável*** se $v \in \mathbb{Z}_n^*$, $j \in \{0, 1, ..., t\}$ e

$$v^{2^j u} \equiv -1 \pmod{n} .$$

Pares aceitáveis certamente existem, já que u é ímpar; podemos escolher $v = n - 1$ e $j = 0$, de modo que $(n - 1, 0)$ é um par aceitável. Agora, escolha o maior j possível tal que exista um par aceitável (v, j) e fixe v de modo que (v, j) seja um par aceitável. Seja

$$B = \{x \in \mathbb{Z}_n^* : x^{2^j u} \equiv \pm 1 \pmod{n}\}.$$

Como B é fechado para multiplicação módulo n, ele é um subgrupo de $\mathbb{Z}_n^*$. Portanto, pelo Teorema 31.15, $|B|$ divide $|\ \mathbb{Z}_n^*\ |$. Toda não testemunha deve ser um membro de B, já que a sequência X produzida por uma não testemunha deve ser toda 1s ou, então, conter um valor -1 não além da j-ésima posição, pela maximalidade de j. (Se (a, j') é aceitável, onde a é uma não testemunha, devemos ter $j' \leq j$, pelo modo como escolhemos j.)

Agora, usamos a existência de v para demonstrar que existe um $w \in \mathbb{Z}_n^* - B$ e, por consequência, que B é um subgrupo próprio de $\mathbb{Z}_n^*$. Como $v^{2^j u} \equiv -1 \pmod{n}$, temos $v^{2^j u} \equiv -1 \pmod{n_1}$ pelo Corolário 31.29 do teorema chinês do resto. Pelo Corolário 31.28, existe um w que satisfaz simultaneamente as equações

$$w \equiv v \pmod{n_1} ,$$
$$w \equiv 1 \pmod{n_2} .$$

Então,

$$w^{2^j u} \equiv -1 \pmod{n_1} ,$$
$$w^{2^j u} \equiv 1 \pmod{n_2} .$$

Pelo Corolário 31.29, $w^{2^j u} \not\equiv 1 \pmod{n_1}$ implica $w^{2^j u} \not\equiv 1 \pmod{n}$, e $w^{2^j u} \not\equiv -1 \pmod{n_2}$ implica $w^{2^j u} \not\equiv -1 \pmod{n}$. Consequentemente, concluímos que $w^{2^j u} \not\equiv \pm 1 \pmod{n}$, e portanto $w \notin B$.

Resta mostrar que $w \in \mathbb{Z}_n^*$, o que fazemos trabalhando primeiro separadamente em módulo n_1 e em módulo n_2. Trabalhando em módulo n_1, observamos que, visto que $v \in \mathbb{Z}_n^*$, temos que $\text{mdc}(v, n) = 1$, e assim também $\text{mdc}(v, n_1) = 1$; se v não tem nenhum divisor comum com n, certamente não tem nenhum divisor comum com n_1. Como $w \equiv v \pmod{n_1}$, vemos que $\text{mdc}(w, n_1) = 1$. Trabalhando em módulo n_2, observamos que $w \equiv 1 \pmod{n_2}$ implica $\text{mdc}(w, n_2) = 1$. Para combinar esses resultados, usamos o Teorema 31.6, que implica que $\text{mdc}(w, n_1 n_2) = \text{mdc}(w, n) = 1$. Isto é, $w \in \mathbb{Z}_n^*$.

Assim, $w \in \mathbb{Z}_n^* - B$, e finalizamos o Caso 2 concluindo que B é um subgrupo próprio de $\mathbb{Z}_n^*$.

Em qualquer dos casos, vemos que o número de testemunhas de que n é composto é pelo menos $(n - 1)/2$. ■

Teorema 31.39

Para qualquer inteiro ímpar $n > 2$ e inteiro positivo s, a probabilidade de MILLER-RABIN(n, s) errar é no máximo 2^{-s}.

Prova Usando o Teorema 31.38 vemos que, se n é composto, então cada execução do laço **for** das linhas 1–4 tem uma probabilidade de no mínimo 1/2 de descobrir uma testemunha x de que n é composto. MILLER-RABIN só cometerá um erro se for tão sem sorte que não descubra uma testemunha de que n é composto em cada uma das s iterações do laço principal. A probabilidade de ocorrer tal sequência é no máximo 2^{-s}.

Se n é primo, MILLER-RABIN sempre retorna PRIME, e se n é composto, a chance de MILLER-RABIN informar PRIME é no máximo 2”s.

Contudo, ao aplicarmos MILLER-RABIN a um inteiro grande n escolhido aleatoriamente, precisamos considerar também a probabilidade anterior de n ser primo, de modo a interpretar corretamente o resultado de MILLER-RABIN. Suponha que fixamos um comprimento de bits e escolhemos aleatoriamente um inteiro n de bits de comprimento para teste de primalidade. Denotamos por A o evento de n ser primo. Pelo teorema dos números primos (Teorema 31.37), a probabilidade de n ser primo é aproximadamente

$$\begin{aligned} \Pr\{A\} &\approx 1/\ln n \\ &\approx 1{,}443/\beta. \end{aligned}$$

Agora, denotamos por B o evento de MILLER-RABIN retornar PRIME. Temos que

Pr $\{B \mid \mathbf{A}\} = 0$ (ou, o que é equivalente, Pr $\{B \mid A\} = 1$) e Pr $\{B \mid A) \leq 2^{-s}$ (ou, o que é equivalente, Pr $\{B \mid A) > 1 - 2^{-s}$).

Mas qual é Pr $\{A|B\}$, a probabilidade de n ser primo, dado que MILLE-RRABIN retornou PRIME? Pela forma alternativa do teorema de Bayes (equação (C.18)) temos

$$\begin{aligned} \Pr\{A|B\} &= \frac{\Pr\{A\}\Pr\{B|A\}}{\Pr\{A\}\Pr\{B|A\} + \Pr\{\bar{A}\}\Pr\{B|\bar{A}\}} \\ &\approx \frac{1}{1 + 2^{-S}(\ln n - 1)}. \end{aligned}$$

Essa probabilidade não ultrapassa 1/2 até s exceder lg(ln n – 1). Intuitivamente, esse mesmo número de tentativas iniciais é necessário só para que a confiança derivada de não termos encontrado uma testemunha de que n é composto supere o viés anterior em favor de n ser composto. Para um número com = 1.024 bits, esse teste inicial requer aproximadamente

$$\begin{aligned} \lg(\ln n - 1) &\approx \lg(\beta/1.443) \\ &\approx 9 \end{aligned}$$

tentativas. De qualquer modo, escolher s = 50 deve ser suficiente para quase toda aplicação imaginável.

Na verdade, a situação é muito melhor. Se estamos tentando encontrar primos grandes aplicando MILLER-RABIN a inteiros ímpares grandes escolhidos aleatoriamente, então é muito improvável que escolher um valor pequeno de s (digamos 3) leve a resultados errôneos, embora não provemos isso aqui. A razão é que, para um inteiro ímpar composto n escolhido aleatoriamente, é provável que o número esperado de não testemunhas de que n é composto seja muito menor que $(n – 1) / 2$.

Entretanto, se o inteiro n não for escolhido aleatoriamente, o melhor que se pode provar é que o número de não testemunhas é no máximo $(n – 1) / 4$, usando uma versão melhorada do Teorema 31.38. Além do mais, existem inteiros n para os quais o número de não testemunhas é $(n – 1) / 4$.

Exercícios

31.8-1 Prove que, se um inteiro ímpar $n > 1$ não é primo ou uma potência prima, existe uma raiz quadrada não trivial de 1 módulo n.

31.8-2 ★ É possível fortalecer ligeiramente o teorema de Euler para a forma

$a^{\lambda(n)} \equiv 1 \pmod n$ para todos $\mathbb{Z}_n^*$,

onde $p_1^{e_1} \cdots p_r^{e_r}$ e $\lambda(n)$ é definido por

$$\lambda(n) = \mathrm{mmc}(\phi(p_1^{e_1}), \cdots \phi(p_r^{e_r})) \qquad (31.42)$$

Prove que $(n) \mid f(n)$. Um número composto n é um número de Carmichael se $(n) \mid n – 1$. O menor número de Carmichael é 561 = 3 · 11 · 17; aqui, (n) = mmc(2, 10, 16) = 80, que é um divisor de 560. Prove que os números de Carmichael devem ser “livres de quadrados” (não divisíveis pelo quadrado de qualquer primo) e o produto de no mínimo, três primos. Por essa razão, eles não são muito comuns.

31.8-3 Prove que, se x é uma raiz quadrada não trivial de 1, módulo n, então mdc(x – 1, n) e mdc(x + 1, n) são divisores não triviais de n.

31.9 ★ FATORAÇÃO DE INTEIROS

Suponha que tenhamos um inteiro n que desejamos ***fatorar***, isto é, decompor esse inteiro em um produto de primos. O teste de primalidade da seção anterior pode nos informar que n é composto, mas não nos informa os fatores primos de n. Fatorar um inteiro grande n parece ser muito mais difícil que simplesmente determinar se n é primo ou composto. Mesmo com os supercomputadores de hoje e com os melhores algoritmos existentes até agora, é inviável fatorar um número arbitrário de 1.024 bits.

Heurística rô de Pollard

É garantido que a tentativa de divisão por todos os inteiros até R fatora completamente qualquer número até R_2. Para a mesma quantidade de trabalho, o procedimento a seguir, Pollard-Rho, fatora qualquer número até R_4 (a menos da má sorte). Como o procedimento é apenas uma heurística, nem seu tempo de execução nem seu sucesso é garantido, embora o procedimento seja muito eficiente na prática. Outra vantagem do procedimento Pollard-Rho é que ele usa somente um número constante de posições de memória. (Se você quisesse, poderia implementar facilmente Pollard-Rho em uma calculadora de bolso programável para determinar fatores de números pequenos.)

```
Pollard-Rho(n)
 1  i = 1
 2  x_1 = Random(0, n − 1)
 3  y = x_1
 4  k = 2
 5  while True
 6      i = i + 1
 7      x_i = (x_{i−1}^2 − 1) mod n
 8      d = mdc(y − x_i, n)
 9      if d ≠ 1 e d ≠ n
10          imprima d
11      if i == k
12          y = x_i
13          k = 2k
```

O procedimento funciona da seguinte maneira: as linhas 1–2 inicializam i como 1 e x_1 como um valor escolhido aleatoriamente em $_n$. O laço **while** que começa na linha 5 itera para sempre, buscando fatores de n. Durante cada iteração do laço **while**, a linha 7 usa a recorrência

$$x_i = \left(x_{i-1}^2 - 1\right) \bmod n \tag{31.43}$$

para produzir o próximo valor de x_i na sequência infinita

$$x_1, x_2, x_3, x_4, \ldots ; \tag{31.44}$$

e a linha 6 incrementa i correspondentemente. Por questão de clareza, o pseudocódigo utiliza variáveis x_i com índices, mas o programa funciona do mesmo modo se todos os índices forem eliminados, já que somente o valor mais recente de x_i precisa ser mantido. Com essa modificação, o procedimento utiliza somente um número constante de posições de memória.

De vez em quando, o programa grava o valor x_i gerado mais recentemente na variável y. Especificamente, os valores que são gravados são aqueles cujos índices são potências de 2:

$$x_1, x_2, x_4, x_8, x_{16}, \ldots ;$$

A linha 3 grava o valor x_i e a linha 12 grava x_k, sempre que i é igual a k. A variável k é inicializada como 2 na linha 4, e a linha 13 dobra esse valor sempre que a linha 12 atualiza y. Então, k segue a sequência 1, 2, 4, 8, ... e sempre dá o índice do próximo valor x_k a ser gravado em y.

As linhas 8–10 tentam encontrar um fator de n usando o valor gravado de y e o valor atual de x_i. Especificamente, a linha 8 calcula o máximo divisor comum $d = \text{mdc}(y - x_i, n)$. A linha 9 determina que d é um divisor não trivial de n, e a

linha 10 imprime *d*.

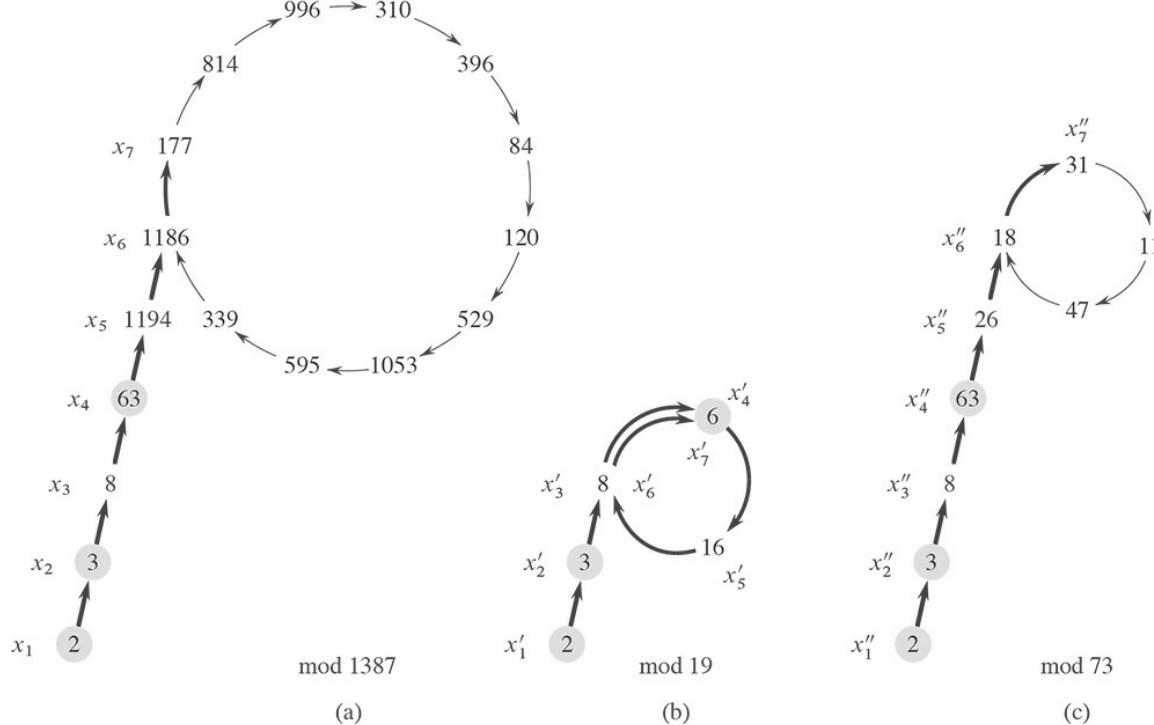
Esse procedimento para encontrar um fator pode parecer um pouco misterioso a princípio. Contudo, observe que POLLARD-RHO nunca imprime uma resposta incorreta; qualquer número que ele imprime é um divisor não trivial de n. Entretanto, POLLARD-RHO poderia não imprimir nada; não há nenhuma garantia de que ele imprimirá algum divisor. Contudo, veremos que há uma boa razão para esperar que POLLARD-RHO imprima um fator p de n depois de $(\sqrt{p})$ iterações do laço **while**. Assim, se n é composto, podemos esperar que esse procedimento descubra divisores suficientes para fatorar n completamente após aproximadamente $n_{1/4}$ atualizações, já que todo fator primo p de n, exceto possivelmente o maior deles, é menor que $\sqrt{n}$.

Iniciamos nossa análise do comportamento desse procedimento estudando o tempo que demora para uma sequência aleatória módulo n repetir um valor. Visto que $_n$ é finito e que cada valor na sequência (31.44) depende apenas do valor anterior, a certa altura a sequência (31.44) se repete. Uma vez alcançado um x_i tal que $x_i = x_j$ para algum $j < i$, estamos em um ciclo, já que $x_i+1 = x_j + 1$, $x_i + 2 = x_j + 2$, e assim por diante. A razão para o nome "heurística rô" é que, como mostra a Figura 31.7, podemos desenhar a sequência $x_1, x_2, ..., x_j - 1$ como a "cauda" da letra grega rô, e o ciclo $x_j, x_j + 1, ..., x_i$ como o "corpo" da letra rô.

Vamos considerar a questão do tempo necessário para que a sequência de x_i se repita. Essa informação não é exatamente o que precisamos, mas veremos mais adiante como modificar o argumento. Para a finalidade dessa estimativa, vamos supor que a função

$$f_n(x) = (x^2 - 1) \bmod n$$

se comporta como uma função "aleatória". É claro que ela não é realmente aleatória, mas considerá-la como tal produz resultados compatíveis com o comportamento observado de POLLARD-RHO. Então podemos supor que cada x_i foi extraído independentemente de $_n$, de acordo com uma distribuição uniforme em $_n$. Pela análise do paradoxo do aniversário da Seção 5.4.1, espe>ramos que sejam executadas $(\sqrt{n})$ etapas antes de a sequência começar a ciclar.

Agora, vamos à modificação requerida. Seja p um fator não trivial de n tal que $\text{mdc}(p, n/p) = 1$. Por exemplo, se a fatoração de $n = p_1^{e_1} p_1^{e_2} ... p_r^{e_r}$, então podemos supor que p é $p_1^{e_1}$. (Se $e_1 = 1$, então p é apenas o menor fator primo de n, um bom exemplo para ter em mente.)

A sequência $\langle x_i \rangle$ induz uma sequência correspondente módulo p, onde

$$x_i' = x_i \bmod p$$

para todo i.

Além disso, como f_n é definido apenas por operações aritméticas (elevação ao quadrado e subtração) módulo n, podemos calcular $x_i' + 1$ a partir de x_i'; a visão "módulo p" da sequência é uma versão menor de que está acontecendo módulo n:

$$
\begin{aligned}
x'_{i+1} &= x_{i+1} \bmod p \\
&= f_n(x_i) \bmod p \\
&= ((x_i^2 - 1) \bmod n) \bmod p \\
&= (x_i^2 \ - 1) \bmod p \qquad\qquad \text{(por Exercício 31.1 − 7)} \\
&= ((x_i \bmod p)^2 - 1) \bmod p \\
&= ((x'_i)^2 - 1) \bmod p \\
&= f_p(x'_i)\cdot
\end{aligned}
$$

Figura 31.7 A heurística rô de Pollard. (**a**) Os valores produzidos pela recorrência $x_i + 1 = (x_i - 1) \bmod 1387$, começando com $x_1 = 2$. A decomposição em fatores primos de 1.387 é 19 · 73. As setas grossas indicam as etapas de iteração que são executadas antes de o fator 19 ser descoberto. As setas finas apontam para valores não alcançados na iteração, para ilustrar a forma "rô" . Os valores sombreados são os valores y armazenados por POLLARD-RHO. O fator 19 é descoberto quando o procedimento alcança $x_7 = 177$, quando mdc(63 - 177, 1.387) = 19 é calculado. O primeiro valor de x que se repetiria é 1.186, mas o fator 19 é descoberto antes de esse valor ser repetido. (**b**) Valores produzidos pela mesma recorrência, módulo 19. Todo valor x_i dado na parte (a) é equivalente, módulo 19, ao valor x'_i mostrado aqui. Por exemplo, tanto $x_4 = 63$ quanto $x_7 = 177$ são equivalentes a 6, módulo 19. (**c**) Valores produzidos pela mesma recorrência, módulo 73. Todo valor x_i dado na parte (a) é equivalente, módulo 73, ao valor x''_i mostrado aqui. Pelo teorema chinês do resto, cada nó na parte (a) corresponde a um par de nós, um da parte (b) e outro da parte (c).

Assim, embora não calculando explicitamente a sequência $\langle x_i' \rangle$, essa sequência é bem definida e obedece à mesma recorrência que a sequência $\langle x_i \rangle$.

Pelo mesmo raciocínio de antes, verificamos que o número esperado de etapas antes de a sequência $\langle x_i' \rangle$ se repetir é $(\sqrt{p})$. Se p é pequeno em comparação com n, a sequência $\langle x_i' \rangle$ poderia se repetir muito mais rapidamente que a sequência $\langle x_i \rangle$. De fato, como as partes (b) e (c) da Figura 31.7 mostram, a sequência $\langle x_i' \rangle$ se repete tão logo os dois elementos da sequência $\langle x_i \rangle$ sejam só equivalentes módulo p, em vez de equivalentes módulo n.

Seja t o índice do primeiro valor repetido na sequência $\langle x_i' \rangle$ e seja $u > 0$ o comprimento do ciclo assim produzido. Isto é, t e $u > 0$ são os menores valores tais que $x'_{t+1} = x'_{t+u+i}$ para todo $i \geq 0$. Pelos argumentos citados, os

valores esperados de t e u são $(\sqrt{p})$. Observe que, se $(x'_{t+1} = x'_{t+u+i})$, então $p \mid (x_{t+u+i} - x_{t+i})$ Assim, mdc $(x_{t+u+i} - x_{t+i}, n) > 1$.

Portanto, tão logo POLLARD-RHO tenha gravado como y qualquer valor x_k tal que $k > t$, então y mod p está sempre no ciclo módulo p. (Se um novo valor for gravado como y, esse valor também estará no ciclo módulo p.) A certa altura, k recebe um valor maior que u, e então o procedimento executa um laço inteiro ao redor do ciclo módulo p, sem mudar o valor de y. Então o procedimento descobre um fator de n quando x_i se defronta com o valor previamente armazenado de y, módulo p, isto é, quando $x_i \equiv y \pmod{p}$.

Podemos presumir que o fator encontrado seja o fator p, embora ocasionalmente um múltiplo de p possa ser descoberto. Como os valores esperados de t e u são $(\sqrt{p})$, o número esperado de etapas necessárias para produzir o fator p é $(\sqrt{p})$.

Esse algoritmo poderia não funcionar exatamente como esperado por duas razões. A primeira é que a análise heurística do tempo de execução não é rigorosa, e é possível que o ciclo de valores, módulo p, seja muito maior que $(\sqrt{p})$. Nesse caso, o algoritmo funciona corretamente, embora muito mais lentamente que o desejado. Na prática, essa questão parece ser irrelevante.

A segunda é que os divisores de n produzidos por esse algoritmo poderiam ser sempre um dos fatores triviais 1 ou n. Por exemplo, suponha que $n = pq$, onde p e q são primos. Pode acontecer de os valores de t e u para p serem idênticos aos valores de t e u para q e, assim, o fator p é sempre revelado na mesma operação mdc que revela o fator q. Visto que ambos os fatores são revelados ao mesmo tempo, o fator trivial $pq = n$ é revelado, o que é inútil. Novamente, esse problema parece ser insignificante na prática. Se necessário, a heurística pode reiniciar com uma recorrência diferente da forma $x_{i+1} = (x_{2i} - c) \bmod n$. (Devemos evitar os valores $c = 0$ e $c = 2$ por razões que não abordaremos aqui, mas os outros valores são bons.)

É claro que essa análise é heurística e não rigorosa, já que a recorrência não é realmente "aleatória". Todavia, o procedimento funciona bem na prática e parece ser tão eficiente quanto essa análise heurística indica. É o método preferido para encontrar fatores primos pequenos de um número grande. Para fatorar completamente um número composto n de bits, basta deter>minar todos os fatores primos menores que $n_{1/2}$ e, portanto, esperamos que POLLARD-RHO exija no máximo $n_{1/2} = 2^{/4}$ operações aritméticas e, no máximo, $n_{1/4}\ ^4 = 2^{/4}\ ^2$ operações com bits. A >capacidade de POLLARD-RHO para encontrar um fator pequeno p de n com um número esperado de $(\sqrt{p})$ operações aritméticas é, muitas vezes, sua característica mais atraente.

Exercícios

31.9-1 Consultando o histórico de execução mostrado na Figura 31.7(a), quando POLLARD-RHO imprime o fator 73 de 1.387?

31.9-2 Suponha que tenhamos uma função $f : _n \to _n$ e um valor inicial $x_0 \in _n$. Defina $x_i = f(x_{i-1})$ para $i = 1, 2, \ldots$ Sejam t e $u > 0$ os menores valores tais que $x_{t+i} = x_{t+u+i}$ para $i = 0, 1, \ldots$.. Na terminologia do algoritmo rô de Pollard, i é o comprimento da cauda e u é o comprimento do ciclo do rô. Dê um algoritmo eficiente para determinar t e u exatamente e analise seu tempo de execução.

31.9-3 Quantas etapas você esperaria que POLLARD-RHO exija para descobrir um fator da forma p_e, em que p seja primo e $e > 1$?

31.9-4* Uma desvantagem de POLLARD-RHO como foi escrito é que ele exige um cálculo de mdc para cada etapa da recorrência. Em vez disso, poderíamos criar um lote de cálculos de mdc acumulando o produto de vários x_i em uma linha e depois usando esse produto em vez de x_i no cálculo do mdc. Descreva cuidadosamente como você implementaria essa ideia, por que ela funciona e que tamanho de lote você escolheria como o mais efetivo ao trabalhar com um número n de bits.

Problemas

31-1 Algoritmo binário de mdc

A maioria dos computadores efetua operações de subtração, teste de paridade (ímpar ou par) de um inteiro binário e divisão por 2 mais rapidamente que o cálculo de restos. Este problema investiga o ***algoritmo de mdc binário***, que evita os cálculos de resto usados no algoritmo de Euclides.

a. Prove que, se a e b são pares, então mdc(a, b) = 2 mdc(a/2, b/2).

b. Prove que, se a é ímpar e b é par, então mdc(a, b) = mdc(a, b/2).

c. Prove que, se a e b são ímpares, mdc(a, b) = mdc((a - b)/2, b).

d. Projete um algoritmo de mdc binário eficiente para inteiros de entrada a e b, onde $a \geq b$, que é executado no tempo $O(\lg a)$. Suponha que cada subtração, teste de paridade e divisão em metades possa ser executada em tempo unitário.

31-2 Análise de operações com bits no algoritmo de Euclides

a. Considere o algoritmo comum de "lápis e papel" para a divisão longa: dividir a por b, >gerando um quociente q e um resto r. Mostre que esse método exige $O((1 + \lg q) \lg b)$ operações com bits.

b. Defina $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Mostre que o número de operações com bits executadas por EUCLID para reduzir o problema de calcular mdc(a, b) ao problema de calcular mdc(b, a mod b) é no máximo $c(m(a, b) - m(b, a \bmod b))$ para alguma constante $c > 0$ suficientemente grande.

c. Mostre que EUCLID(a, b) exige $O(m(a, b))$ operações com bits em geral e $O(_2)$ operações com bits quando aplicado a duas entradas de bits.

31-3 Três algoritmos para números de Fibonacci

Este problema compara a eficiência de três métodos para calcular o n-ésimo número de Fibonacci F_n, dado n. Suponha que o custo de adicionar, subtrair ou multiplicar dois números seja $O(1)$, independentemente do tamanho dos números.

a. Mostre que o tempo de execução do método recursivo direto para calcular F_n baseado na recorrência (3.22) é exponencial em n. (Veja, por exemplo, o procedimento FIB na página 775.)

b. Mostre como calcular F_n no tempo $O(n)$ usando memoização.

c. Mostre como calcular F_n no tempo $O(\lg n)$ usando apenas adição e multiplicação de inteiros. (*Sugestão*: Considere a matriz

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

e suas potências.)

d. Agora, suponha que somar dois números de bits demora o tempo () e que multiplicar dois números de bits demora o tempo (2). Qual é o tempo de execução desses três métodos sob essa medida de custo mais razoável para as operações aritméticas elementares?

31-4 Resíduos quadráticos

Seja p um primo ímpar. Um número $a \in {}^*_n$ é um ***resíduo quadrático*** se a equação $x_2 = a \pmod p$ tem uma solução para a incógnita x.

a. Mostre que existem exatamente $(p - 1)/2$ resíduos quadráticos, módulo p.

b. Se p é primo, definimos o ***símbolo de Legendre*** $\left(\frac{a}{p}\right)$ para $a \in \mathbb{Z}_n^*$ como 1 se a é um resíduo quadrático módulo p e -1 em caso contrário. Prove que, se $a \in \mathbb{Z}_n^*$, então

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p .$$

Dê um algoritmo eficiente que determine se um dado número a é ou não um resíduo quadrático módulo p. Analise a eficiência de seu algoritmo.

c. Prove que, se p é um primo da forma $4k + 3$ e a é um resíduo quadrático em $^*p$, então $a_k + 1$ mod p é uma raiz quadrada de a, módulo p. Quanto tempo é necessário para determinar a raiz quadrada de um resíduo quadrático a módulo p?

d. Descreva um algoritmo aleatorizado eficiente para determinar um resíduo não quadrático, módulo primo p arbitrário, isto é, um membro de $^*p$ que não é um resíduo quadrático. Quantas operações aritméticas seu algoritmo exige em média?

NOTAS DO CAPÍTULO

Niven e Zuckerman [265] dão uma excelente introdução à teoria elementar dos números. Knuth [210] contém uma boa discussão de algoritmos para encontrar o máximo divisor comum, bem como outros algoritmos básicos da teoria dos números. Bach [30] e Riesel [295] apresentam levantamentos mais recentes de teoria computacional dos números. Dixon [92] apresenta uma visão geral da fatoração e do teste de primalidade. Os procedimentos de conferências editados por Pomerance [280] contêm várias resenhas interessantes. Mais recentemente, Bach e Shallit [31] apresentaram uma cobertura excepcional dos conceitos básicos da teoria computacional dos números.

Knuth [210] discute a origem do algoritmo de Euclides. Ele aparece no Livro 7, Proposições 1 e 2, de *Elementos* do matemático grego Euclides, que foi escrito em torno de 300 a.C. A descrição de Euclides pode ter sido derivada de um algoritmo criado por Eudoxus, por volta de 375 a.C. O algoritmo de Euclides pode ostentar a honra de ser o mais antigo algoritmo não trivial; ele só encontra rival em um algoritmo para multiplicação conhecido pelos antigos egípcios. Shallit [312] narra a história da análise do algoritmo de Euclides.

Knuth atribui um caso especial do teorema chinês do resto (Teorema 31.27) ao matemático chinês Sun-Tsu, que viveu em alguma época entre 200 a.C. e 200 d.C. – a data é bastante incerta. O mesmo caso especial foi apresentado pelo matemático grego Nichomachus por volta de 100 d.C., generalizado por Chin Chiu-Shao em 1247. O teorema chinês do resto foi finalmente enunciado e provado em sua total generalidade por L. Euler, em 1734.

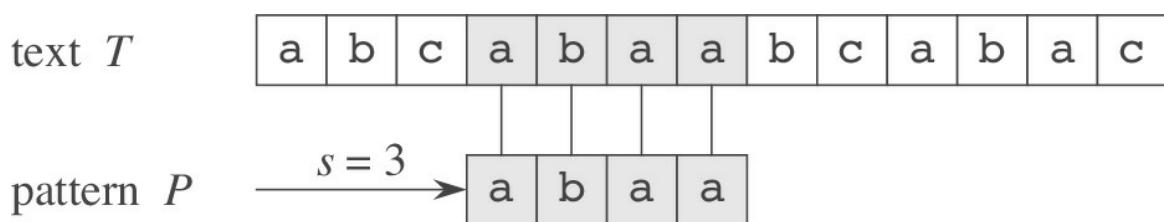
O algoritmo aleatorizado do teste de primalidade apresentado aqui se deve a Miller [225] e Rabin [289]; é o algoritmo aleatorizado para teste de primalidade mais rápido que se conhece, a menos de fatores constantes. A prova do Teorema 31.39 é uma ligeira adaptação de uma sugerida por Bach [29]. Uma prova de um resultado mais forte para

MILLER-RABIN foi dada por Monier [258, 259]. A aleatorização parece ser necessária para obter um algoritmo de teste de primalidade de tempo polinomial. O algoritmo determinístico mais rápido conhecido para teste de primalidade é a versão de Cohen-Lenstra [75] do teste de primalidade de Adleman, Pomerance e Rumely [3]. Ao testar a primalidade de um número *n* de comprimento é lg(*n* + 1)ù , ele é executado no tempo $(\lg n)^{O(\lg \lg \lg n)}$, que é apenas ligeiramente superpolinomial.

O problema de encontrar primos grandes "aleatórios" é discutido agradavelmente em um artigo de Beauchemin, Brassard, Crépeau, Gutier e Pomerance [36].

O conceito de um sistema de criptografia de chave pública se deve a Diffie e Hellman [88]. O sistema de criptografia RSA foi proposto em 1977 por Rivest, Shamir e Adleman [296]. Desde então, a área da criptografia floresceu. Nosso entendimento do sistema de criptografia SA se aprofundou, e implementações modernas utilizam refinamentos significativos das técnicas básicas apresentadas aqui. Além disso, muitas novas técnicas foram desenvolvidas para demonstrar que sistemas de criptografia são seguros. Por exemplo, Goldwasser e Micali [142] mostram que a aleatorização poder ser uma ferramenta efetiva no projeto de esquemas confiáveis de criptografia de chave pública. Para esquemas de assinaturas, Goldwasser, Micali e Rivest [143] apresentam um esquema de assinatura digital para o qual todo tipo concebível de falsificação é comprovadamente tão difícil quanto a decomposição em fatores primos. Menezes, van Oorschot, e Vanstone [254] apresentam uma visão geral da criptografia aplicada.

A heurística rô para fatoração de inteiros foi criada por Pollard [277]. A versão apresentada aqui é uma variante proposta por Brent [57].

Os melhores algoritmos para fatorar números grandes têm um tempo de execução que cresce de forma aproximadamente exponencial com a raiz cúbica do comprimento do número *n* a ser fatorado. O algoritmo geral de fatoração de peneira de corpo numérico foi desenvolvido por Buhler, Lenstra e Pomerance [58] como uma extensão das idéias sobre o algoritmo de fatoração de peneira de corpo numérico criado por Pollard [278] e Lenstra *et al.* [232] e refinado por Coppersmith [78] e outros é talvez o algoritmo mais eficiente em geral para grandes entradas. Embora seja difícil apresentar uma análise rigorosa desse algoritmo, sob hipóteses razoáveis podemos deduzir uma estimativa de tempo de execução igual a $L(1/3, n)^{1,902 + o(1)}$, onde $L(\alpha, n) = e^{(\ln n)^a (\ln \ln n)^{1 - a}}$.

O método de curva elíptica criado por Lenstra [233] pode ser mais efetivo para algumas entradas que o método de crivo de corpo de números já que, como o método rô de Pollard, ele pode encontrar um pequeno fator primo *p* com bastante rapidez. Por esse método, o tempo para >encontrar *p* é estimado em $L(1/2, p)^{\sqrt{2}+0(1)}$.

32 CORRESPONDÊNCIA DE CADEIAS

Frequentemente, os programas de edição de textos precisam encontrar todas as ocorrências de um padrão no texto. Em geral, o texto é um documento que está sendo editado, e o padrão procurado é uma palavra específica fornecida pelo usuário. Algoritmos eficientes para esse problema — denominados "algoritmos de correspondência de cadeias" — podem ajudar muito no nível de resposta do programa de edição de textos. Entre suas muitas outras aplicações, algoritmos de correspondência de cadeias procuram padrões particulares em sequências de DNA. Programas de busca na Internet também os usam para achar páginas relevantes para as consultas.

Formalizamos o problema de correspondência de cadeias da maneira mostrada a seguir. Supomos que o texto seja um arranjo $T[1 .. n]$ de comprimento n e que o padrão seja um arranjo $P[1..m]$ de comprimento $m \leq n$. Supomos ainda que os elementos de P e T sejam caracteres extraídos de um alfabeto finito S. Por exemplo, podemos ter $S = \{0, 1\}$ ou $S = \{a, b, ..., z\}$. Os arranjos de caracteres P e T são frequentemente denominados ***cadeias*** de caracteres.

Referindo-nos à Figura 32.1, dizemos que o padrão P ***ocorre com deslocamento s*** no texto T (ou, o que é equivalente, que o padrão P ***ocorre começando na posição s* + 1** no texto T) se $0 \leq s \leq n - m$ e $T[s + 1 .. s + m] = P[1 .. m]$ (isto é, se $T[s + j] = P[j]$, para $1 \leq j \leq m$). Se P ocorre com deslocamento s em T, então denominamos s ***deslocamento válido***; caso contrário, denominamos s ***deslocamento inválido***. O ***problema da correspondência de cadeias*** é o problema de encontrar todos os deslocamentos válidos com os quais um determinado padrão P ocorre em dado texto T.

Com exceção do algoritmo ingênuo de força bruta, que examinamos na Seção 32.1, cada algoritmo de correspondência de cadeias neste capítulo executa algum pré-processamento baseado no padrão e depois encontra todos os deslocamentos válidos; denominamos essa fase posterior "correspondência". A Figura 32.2 mostra os tempos de pré-processamento e correspondência para cada um dos algoritmos neste capítulo. O tempo de execução total de cada algoritmo é a soma dos tempos de pré-processamento e correspondência. A Seção 32.2 apresenta um interessante algoritmo de correspondências de cadeias, criado por Rabin e Karp. Embora o tempo de execução do pior caso $Q((n - m + 1)m)$ desse algoritmo não seja melhor que o do método ingênuo, ele funciona muito melhor em média, na prática. O algoritmo também pode produzir generalizações interessantes para outros problemas de correspondência de padrões. A Seção 32.3 descreve um algoritmo de correspondência de cadeias que começa pela construção de um autômato finito projetado especificamente para procurar em um texto ocorrências do >padrão P dado. O tempo de pré-processamento desse algoritmo é $O(m|S|)$, mas o tempo de correspondência é somente $Q(n)$. A Seção 32.4 apresenta o algoritmo semelhante, porém muito mais inteligente, de Knuth-Morris-Pratt (ou KMP). Esse algoritmo tem o mesmo tempo de correspondência $Q(n)$ e reduz o tempo de pré-processamento a apenas $Q(m)$.

Figura 32.1 Exemplo do problema da correspondência de cadeias, no qual queremos encontrar todas as ocorrências do padrão P = abaa no texto T abcabaabcabac . O padrão ocorre apenas uma vez no texto, no deslocamento $s = 3$, que denominamos deslocamento válido. Uma linha vertical liga cada caractere do padrão a seu caractere correspondente no texto, e todos os caracteres para os quais ocorreu a correspondência estão sombreados.

Notação e terminologia

Denotamos por S* (lê-se "sigma estrela") o conjunto de todas as cadeias de comprimento finito extraídas do alfabeto S. Neste capítulo, consideramos somente cadeias de comprimento finito. A ***cadeia vazia*** de comprimento zero, denotada por , também pertence a S*. O comprimento de uma cadeia x é denotado por $|x|$. A ***concatenação*** de duas cadeias x e y, representada por xy, tem comprimento $|x| + |y|$ e consiste nos caracteres de x seguidos pelos caracteres de y.

Dizemos que uma cadeia w é um ***prefixo*** de uma cadeia x, denotada por w x, se $x = wy$ >para alguma cadeia $y \in$ S*. Observe que, se w x, então $|w| \leq |x|$. De modo semelhante, dize>mos que uma cadeia w é um ***sufixo*** de uma cadeia x, representada por se x yw para algum $y \in$ S*. Do mesmo modo que para um prefixo, w x implica $|w| \leq |x|$. Por exemplo, temos ab abcca e cca abcca. A cadeia vazia é um sufixo e também um prefixo para todas as cadeias. Para quaisquer cadeias x e y e qualquer caractere a, temos x y se e somente se xa ya. Observe também que e são relações transitivas. O lema a seguir será útil mais adiante.

Lema 32.1 (Lema dos sufixos sobrepostos)

Suponha que x, y e z sejam cadeias tais que x z e y z. Se $|x| \leq |y|$, então x y. Se $|x| \geq |y|$, então y x. Se $|x| = |y|$, então $x = y$.

Prova Consulte a Figura 32.2 para ver uma prova gráfica.

Para abreviar a notação, denotamos o prefixo de k caracteres $P[1..k]$ do padrão $P[1..m]$ por P_k. Assim, $P_0 =$ e $P_m = P = P[1..m]$. De modo semelhante, denotamos o prefixo de k caracteres do texto T por T_k. Usando essa notação, podemos enunciar o problema da correspondência de cadeias como o de encontrar todos os deslocamentos s no intervalo $0 \leq s \leq n - m$ tais que P $T_s + m$.

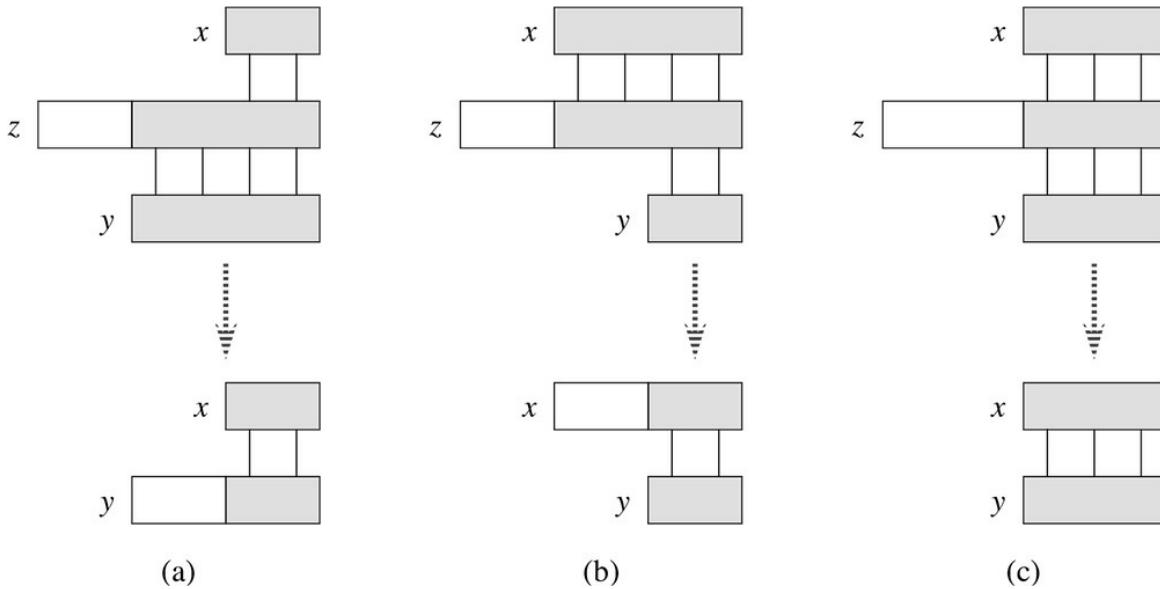
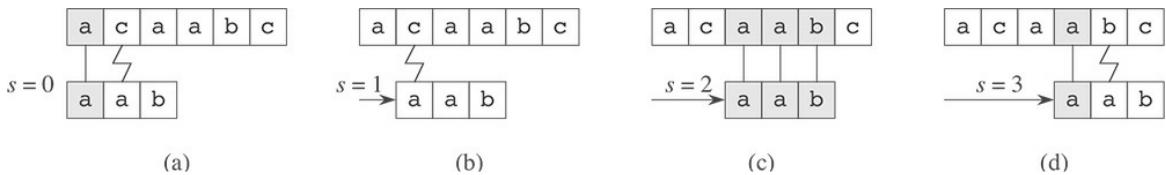
Em nosso pseudocódigo, permitimos que a comparação para determinação da igualdade de duas cadeias de comprimentos iguais seja uma operação primitiva. Se as cadeias são comparadas da esquerda para a direita e a comparação parar quando for descoberta uma incompatibilidade, supomos que o tempo despendido por tal teste é função linear do número de caracteres correspondentes descobertos. Para sermos precisos, supomos que o teste "$x == y$" demora o tempo Q(t + 1), onde t é o comprimento da mais longa cadeia z tal que z x e z y. (Escrevemos Q(t + 1) em vez de Q(t) para tratar o caso no qual $t = 0$; os primeiros caracteres comparados não são correspondentes, mas fazer essa comparação demora uma quantidade de tempo positiva.)

Algoritmo	Tempo de pré-processamento	Tempo de correspondência
Ingênuo	0	$O((n - m + 1)m)$
Rabin-Karp	$\Theta(m)$	$O((n - m + 1)m)$
Autômato finito	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Figura 32.2 Os algoritmos de correspondência de cadeias neste capítulo e seus tempos de pré-processamento e correspondência.

Figura 32.3 Uma prova gráfica do Lema 32.1. Supomos que x z e y z. As três partes da figura ilustram os três casos do lema. Linhas verticais ligam regiões correspondentes (sombreadas) das cadeias. (**a**) Se $|x|$ $|y|$, então x y. (**b**) Se $|x|$ $|y|$, então y x. (**c**) Se $|x| = |y|$, então $x = y$.

32.1 O ALGORITMO INGÊNUO DE CORRESPONDÊNCIA DE CADEIAS

O algoritmo ingênuo encontra todos os deslocamentos válidos usando um laço que verifica a condição $P[1 \mathinner{..} m] = T[s + 1 \mathinner{..} s + m]$ para cada um dos $n - m + 1$ valores possíveis de s.

```
NAIVE-STRING-MATCHER(T, P)
1  n = T. comprimento
2  m = P. comprimento
3  for s = 0 to n – m
4        if P[1 .. m] == T[s + 1 .. s + m]
5        imprimir "Padrão ocorre com deslocamento" s
```

A Figura 32.4 retrata o procedimento ingênuo de procurar a correspondência como fazer deslizar sobre o texto um "gabarito" que contém o padrão e observar para quais deslocamentos todos os caracteres no gabarito são iguais aos caracteres correspondentes no texto. O laço **for** nas linhas 3–5 considera cada deslocamento possível explicitamente. O teste na linha 4 determina se o deslocamento em questão é válido ou não; esse teste executa o laço implicitamente para verificar posições de caracteres correspondentes até que todas as posições correspondam ou até ser encontrada uma incompatibilidade. A linha 5 imprime cada deslocamento s válido.

O procedimento Naive-String-Matcher demora o tempo $O((n - m + 1)m)$, e esse limite é justo no pior caso. Por exemplo, considere a cadeia de texto a_n (uma cadeia de n caracteres a) e o padrão a^m. Para cada um dos $n - m + 1$ valores possíveis do deslocamento s, o laço implícito na linha 4 para comparar caracteres correspondentes deve ser executado m vezes para validar o deslocamento. Assim, o tempo de execução do pior caso é $Q((n - m + 1)m)$, que é $Q(n_2)$ se $m = n/2$. Como não requer nenhum pré-processamento, o tempo de execução de Naive-String-Matcher é igual ao seu tempo de correspondência.

Figura 32.4 A operação do algoritmo de correspondência de cadeias ingênuo para o padrão P = aab e o texto T = acaabc. Podemos imaginar o padrão P como um gabarito que fazemos deslizar próximo ao texto. **(a)–(d)** Os quatro alinhamentos sucessivos tentados pelo algoritmo de correspondência de cadeias ingênuo. Em cada parte, linhas verticais ligam regiões correspondentes que coincidem com o gabarito (sombreadas), e uma linha quebrada liga o primeiro caractere incompatível, se houver. O algoritmo encontra uma ocorrência do padrão, no deslocamento $s = 2$, mostrada na parte **(c)**.

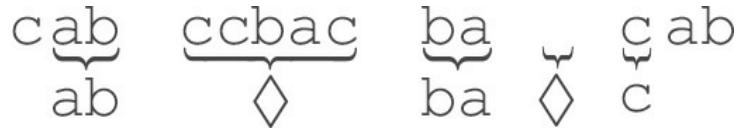
Como veremos, Naive-String-Matcher não é um procedimento ótimo para esse problema. Na realidade, neste capítulo veremos que o algoritmo Knuth-Morris-Pratt é muito melhor para o pior caso. O matcher de cadeias ingênuo é ineficiente porque ignora inteiramente informações adquiridas do texto para um valor de s quando considera outros valores de s. Porém, tais informações podem ser muito valiosas. Por exemplo, se P = aaab e descobrirmos que $s = 0$ é válido, então nenhum dos deslocamentos 1, 2 ou 3 é válido, já que $T[4]$ = b. Nas próximas seções, examinaremos diversas maneiras de fazer uso efetivo desse tipo de informação.

Exercícios

32.1-1 Mostre as comparações que o matcher de cadeias ingênuo realiza para o padrão P = 0001 no texto T = 000010001010001.

32.1-2 Suponha que todos os caracteres no padrão P sejam diferentes. Mostre como acelerar Naive-String-Matcher para ser executado no tempo $O(n)$ em um texto T de n caracteres.

32.1-3 Suponha que o padrão P e o texto T sejam cadeias de comprimento m e n, respectivamente, escolhidas *aleatoriamente* de um alfabeto d-ário $S_d = \{0, 1, ..., d - 1\}$, onde $d \geq 2$.

Mostre que o número *esperado* de comparações de caractere para caractere feitas pelo laço implícito na linha 4 do algoritmo ingênuo é

$$(n - m + 1)\ \frac{1 - d^{-m}}{1 - d^{-1}}\ \leq 2((n - m + 1)$$

para todas as execuções desse laço. (Suponha que o ingênuo interrompe a comparação de caracteres para um dado deslocamento, uma vez encontrada uma incompatibilidade ou se o padrão inteiro coincidir.) Assim, para cadeias escolhidas aleatoriamente, o algoritmo ingênuo é bastante eficiente.

32.1-4 Suponha que permitimos que o padrão P contenha ocorrências de um ***caractere lacuna*** ◊ que pode corresponder a uma cadeia de caracteres *arbitrária* (mesmo uma cadeia de comprimento zero). Por exemplo, o padrão ab◊ba◊c ocorre no texto como cabccbacbacab

e como

```
c ab   ccbac   ba     c ab
  ab     ◊     ba  ◊  c
```

Observe que o caractere lacuna pode ocorrer um número arbitrário de vezes no padrão, mas de modo algum no texto. Dê um algoritmo de tempo polinomial para determinar se tal padrão P ocorre em um dado texto T e analise o tempo de execução de seu algoritmo.

32.2 O ALGORITMO RABIN-KARP

Rabin e Karp propuseram um algoritmo de correspondência de cadeias que funciona bem na prática e que também pode ser generalizado para outros algoritmos para problemas relacionados, como o da correspondência de padrões bidimensionais. O algoritmo Rabin-Karp usa o tempo de pré-processamento Q(m), e seu tempo de execução do pior caso é Q(($n - m + 1$)m). Todavia, com base em certas hipóteses, seu tempo de execução do caso médio é melhor.

Esse algoritmo faz uso de noções elementares da teoria dos números, como a equivalência de dois números módulo um terceiro número. Seria interessante consultar a Seção 31.1, que apresenta as definições relevantes.

Para fins de explanação, vamos supor que S = {0,1,2,...,9}, de modo que cada caractere seja um dígito decimal. (No caso geral, podemos supor que cada caractere seja um dígito em base d, onde d = |S|.) Então, podemos ver uma cadeia de k caracteres consecutivos como a representação de um número decimal de comprimento k. Assim, a cadeia de caracteres 31415 corresponde ao número decimal 31.415. Como interpretamos os caracteres de entrada como símbolos gráficos e dígitos, nesta seção achamos que é conveniente denotá-los como denotaríamos dígitos, em nossa fonte de texto padrão.

Dado um padrão $P[1..m]$, seja p seu valor decimal correspondente. De modo semelhante, dado um texto $T[1..n]$, seja t_s o valor decimal da subcadeia de comprimento m $T[s + 1..s + m]$, para $s = 0, 1, ..., n - m$. Certamente, $t_s = p$ se e somente se $T[s + 1..s + m] = P[1..m]$; assim, s é um deslocamento válido se e somente se $t_s = p$. Se pudéssemos calcular p no tempo Q(m) e todos os valores t_s em um tempo total Q($n - m + 1$),[1] poderíamos determinar todos os deslocamentos válidos s no tempo Q(m) + Q($n - m + 1$) = Q(n), comparando p com cada um dos valores t_s. (Por enquanto, não vamos nos preocupar com a possibilidade de que p e os valores de t_s possam ser números muito grandes.)

Podemos calcular p no tempo Q(m) usando a regra de Horner (veja a Seção 30.1):

$$p = P[m] + 10\,(P[m-1] + 10(P[m-2] + ... + 10(P[2] + 10P[1]) ...)) .$$

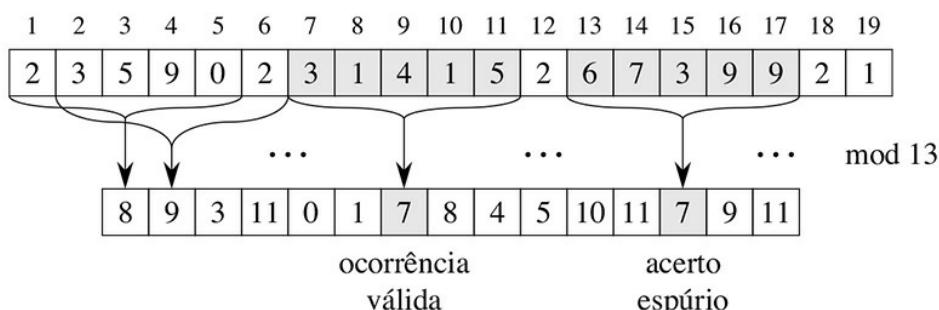
De modo semelhante, podemos calcular t_0 por $T[1..m]$ no tempo Q(m).

Para calcular os valores restantes $t_1, t_2, ..., t_{n-m}$ no tempo Q($n - m$), observe que podemos calcular t_{s+1} a partir de t_s em tempo constante, já que

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] . \tag{32.1}$$

Subtrair $10^{m-1}T[s + 1]$ elimina o dígito de ordem mais alta de t , multiplicar o resultado por 10 desloca o número uma posição de dígito para a esquerda e somar $T[s + m + 1]$ introduz o dígito de ordem baixa adequado. Por exemplo, se $m = 5$ e $t_s = 31415$, desejamos eliminar o dígito de ordem mais alta $T[s + 1] = 3$ e introduzir o novo dígito de ordem baixa (suponha que ele seja $T[s + 5 + 1] = 2$) para obter

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\ &= 14152 . \end{aligned}$$

Se calcularmos a constante 10^{m-1} com antecedência (o que pode ser feito no tempo $O(\lg m)$ usando as técnicas da Seção 31.6, embora para essa aplicação um método direto de $O(m)$ seja suficiente), cada execução da equação

(32.1) toma um número constante de operações aritméticas. Assim, podemos calcular p no tempo Q(m) e calcular todos os t_0, t_1, ..., t_{n-m} no tempo Q($n - m + 1$). Portanto, podemos encontrar todas as ocorrências do padrão P[1..m] no texto T[1..n] com o tempo de pré-processamento Q(m) e o tempo de correspondência Q($n - m + 1$).

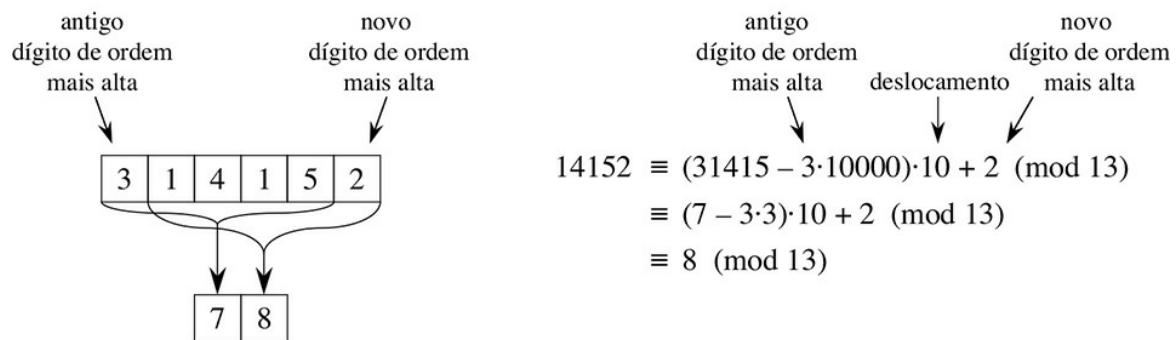
Até agora ignoramos intencionalmente um problema: p e t_s podem ser demasiadamente grandes para que possamos trabalhar com eles de uma forma conveniente. Se P contém m caracteres, não seria razoável considerar que cada operação aritmética para p (que tem m dígitos de comprimento) demora um "tempo constante". Felizmente, é fácil resolver esse problema, como mostra a Figura 32.5: calcule p e os valores t_s módulo um módulo adequado q. Podemos calcular p módulo p_0 no tempo Q(m) e todos os valores t_s módulo q no tempo Q($n - m + 1$). Se escolhermos o módulo q como um primo tal que $10q$ caiba em uma palavra de computador, poderemos executar todos os cálculos necessários com aritmética de precisão simples. Em geral, com um alfabeto d-ário {0, 1, ... $d - 1$}, escolhemos q de modo que dq caiba em uma palavra de computador e ajustamos a equação de recorrência (32.1) para calcular com módulo q; assim, ela se torna

$$t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1] \bmod q \,. \tag{32.2}$$

Figura 32.5 O algoritmo Rabin-Karp. Cada caractere é um dígito decimal, e calculamos valores módulo 13. (**a**) Uma cadeia de texto. Uma janela de comprimento 5 está sombreada. O valor numérico do número sombreado, calculado módulo 13, produz o valor 7. (**b**) A mesma

cadeia de texto com valores calculados módulo 13 para cada posição possível de uma janela de comprimento 5. Considerando o padrão P = 31415, procuramos janelas cujo valor módulo 13 seja 7, já que 31415 ≡ 7 (mod 13). O algoritmo encontra duas dessas janelas, sombreadas na figura. A primeira, que começa na posição de texto 7, é de fato uma ocorrência do padrão, enquanto a segunda, que começa na posição de texto 13, é um acerto espúrio. (**c**) Como calcular o valor para uma janela em tempo constante, dado o valor para a janela anterior. A primeira janela tem valor 31415. Descartando o dígito 3 de ordem mais alta, deslocando-se para a esquerda (multiplicando por 10) e depois somando o dígito de ordem baixa 2 resulta o novo valor 14152. Como todos os cálculos são executados módulo 13, o valor para a primeira janela é 7 e o valor calculado para a nova janela é 8.

---

onde $h \equiv d^{m-1} \pmod q$ é o valor do dígito "1" na posição de ordem alta de uma janela de texto de m dígitos.

Entretanto, a solução para trabalhar com módulo q não é perfeita: $t_s \equiv p \pmod q$ não implica que $t_s = p$. Por outro lado, se $t_s \not\equiv p \pmod q$, então definitivamente temos que $t_s \neq p$, de modo que o deslocamento s é não válido. Assim, podemos usar o teste $t_s \equiv p \pmod q$ como um teste heurístico rápido para eliminar deslocamentos s não válidos. Qualquer deslocamento s para o qual $t_s \equiv p \pmod q$, deve passar por um teste adicional para verificar se s é realmente válido ou se temos apenas um ***acerto espúrio***. Esse teste adicional verifica explicitamente a condição $P[1 \,..\, m] = T[s + 1 \,..\, s + m]$. Se q é suficientemente grande, esperamos que a ocorrência de acertos espúrios seja infrequente o suficiente para que o custo da verificação extra seja baixo.

O procedimento a seguir torna essas ideias precisas. As entradas para o procedimento são o texto T, o padrão P, a base d a utilizar (que em geral é considerada $|\Sigma|$) e o primo q a empregar.

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n = T. comprimento
 2  m = P. comprimento
 3  h = d^(m-1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                  // pré-processamento
 7      p = (dp + P[i]) mod q
 8      t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n - m    // correspondência
10      if p == t_s
11      if P[1..m] = T[s + 1..s + m]
12          imprimia "Padrão ocorre com deslocamento" s
13      if s < n - m
14          t_(s+1) = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

O procedimento Rabin-Karp-Matcher funciona da seguinte maneira: todos os caracteres são interpretados como dígitos em base d. Os índices em t são fornecidos apenas por clareza; o programa funciona corretamente se todos os índices forem descartados. A linha 3 inicializa h com o valor da posição de dígito de ordem mais alta de uma janela de m dígitos. As linhas 4–8 calculam p como o valor de $P[1...m] \bmod q$ e t_0 como o valor de $T[1 \,..\, m] \bmod q$. O laço **for** das linhas 9–14 itera por todos os deslocamentos s possíveis, mantendo o seguinte invariante:

Sempre que a linha 10 é executada, $t_s = T[s + 1 \,..\, s + m] \bmod q$.

Se $p = t_s$ na linha 10 (um "acerto"), então a linha 11 verifica se $P[1..m] = T[s + 1..s + m]$ para eliminar a possibilidade de um acerto espúrio. A linha 12 imprime quaisquer deslocamentos válidos encontrados. Se $s < n - m$ (verificado na linha 13), então o laço **for** será executado no mínimo mais uma vez e, assim, a linha 14 é executada antes para garantir que o invariante de laço será válido quando voltarmos para a linha 10. A linha 14 calcula o valor de $t_{s+1} \bmod q$ pelo valor de $t_s \bmod q$ em tempo constante, usando a equação (32.2) diretamente.

Rabin-Karp-Matcher demora o tempo de processamento Q(m) e o tempo de correspondência é Q(($n - m + 1)m$) no pior caso, já que (como no algoritmo ingênuo de correspondência de cadeias) o algoritmo de Rabin-Karp verifica explicitamente todo deslocamento válido. Se $P = a^m$ e $T = a_n$, então a verificação demora o tempo Q(($n - m + 1)m$), visto que cada um dos $n - m + 1$ deslocamentos possíveis é válido.

Em muitas aplicações, esperamos alguns deslocamentos válidos (talvez alguma constante c deles). Nessas aplicações, o tempo de correspondência esperado do algoritmo é apenas $O((n - m + 1) + cm) = O(n + m)$, mais o tempo necessário para processar acertos espúrios. Podemos basear uma análise heurística na seguinte hipótese: a redução dos valores módulo q age como um mapeamento aleatório de S* para q. (Veja a discussão sobre o uso da divisão para hashing na Seção 11.3.1. É difícil formalizar e provar tal hipótese, embora uma abordagem viável seja supor que q é escolhido aleatoriamente entre inteiros de tamanho adequado. Não buscaremos essa formalização aqui.) Então, podemos esperar que o número de acertos espúrios seja $O(n/q)$, já que podemos estimar que a chance de que um t_s arbitrário será equivalente a p, módulo q, é $1/q$. Visto que há $O(n)$ posições nas quais o teste da linha 10 falha e gastamos o tempo $O(m)$ para cada acerto, o tempo de correspondência do algoritmo Rabin-Karp é

$$O(n) + O(m(v + n/q)) ,$$

onde v é o número de deslocamentos válidos. Esse tempo de execução é $O(n)$ se $v = O(1)$ e escolhemos $q \geq m$. Isto é, se o número esperado de deslocamentos válidos é pequeno ($O(1)$) e escolhermos o primo q maior que o comprimento do padrão, então podemos esperar que o procedimento de Rabin-Karp use somente tempo de correspondência $O(n + m)$. Como $m \leq n$, esse tempo de correspondência esperado é $O(n)$.

Exercícios

32.2-1 Trabalhando com módulo $q = 11$, quantos acertos espúrios o procedimento de Rabin-karp matcher encontra no texto $T = 3141592653589793$ ao procurar o padrão $P = 26$?

32.2-2 Como você estenderia o método Rabin-Karp ao problema de examinar uma cadeia de texto em busca de uma ocorrência de qualquer padrão de um dado conjunto de k padrões? Comece supondo que todos os k padrões têm o mesmo comprimento. Então generalize sua solução para permitir que os padrões tenham comprimentos diferentes.

32.2-3 Mostre como estender o método Rabin-Karp para tratar o problema de procurar por um padrão $m \cdot m$ dado em um arranjo de caracteres $n \cdot n$. (O padrão pode ser deslocado na vertical e na horizontal, mas não pode ser girado.)

32.2-4 Alice tem uma cópia de um longo arquivo de n bits $A = \langle a_{n-1}, a_{n-2}, ..., a_0 \rangle$ e Bob tem um outro arquivo de n bits $B = \langle b_{n-1}, b_{n-2}, ..., b_0 \rangle$. Alice e Bob desejam saber se seus arquivos são idênticos. Para evitar transmitir os arquivos A ou B inteiros, eles usam a seguinte verificação probabilística. Juntos, os dois selecionam um primo $q > 1000n$ e selecionam aleatoriamente um inteiro x de $\{0, 1, ... q - 1\}$. Então, Alice avalia

$$A(x) = \left(\sum_{i=0}^{n-1} a_i x^i \right) \bmod q$$

e Bob também avalia $B(x)$. Prove que, se $A \neq B$, existe no máximo uma chance em 1.000 de que $A(x) = B(x)$; por outro lado, se os dois arquivos forem iguais, $A(x)$ é necessariamente igual a $B(x)$. (*Sugestão*: Veja o Exercício 31.4-4.)

32.3 CORRESPONDÊNCIA DE CADEIAS COM AUTÔMATOS FINITOS

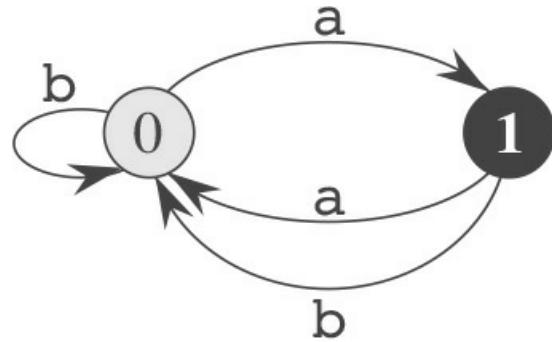
Muitos algoritmos de correspondência de cadeias constroem um autômato finito — uma máquina simples para processar informações — que varre a cadeia de texto T em busca de todas as ocorrências do padrão P. Esta seção apresenta um método para construir tal autômato. Esses autômatos de correspondência de cadeias são muito eficientes: examinam cada caractere de texto *exatamente uma vez* e demoram tempo constante por caractere de texto. Portanto, o tempo de correspondência usado — após o pré-processamento do padrão para construir o autômato — é Q(n). Porém, o tempo para construir o autômato pode ser grande, se S é grande. A Seção 32.4 descreve um modo esperto de contornar esse problema.

Começamos esta seção com a definição de autômato finito. Em seguida, examinamos um autômato especial de correspondência de cadeias e mostramos como usá-lo para encontrar ocorrências de um padrão em um texto. Finalmente, mostraremos como construir o autômato de correspondência de cadeias para um padrão de entrada dado.

Autômatos finitos

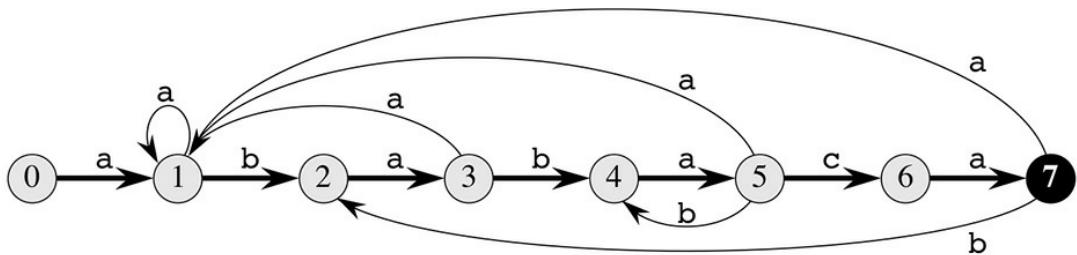
Um ***autômato finito*** M, ilustrado na Figura 32.6, é uma 5-tupla (Q, q_0, A, S, d), onde

- Q é um conjunto finito de ***estados***,
- $q_0 \in Q$ é o ***estado inicial***,
- $A \subseteq Q$ é um conjunto distinto de ***estados aceitadores***,
- S é um ***alfabeto de entrada*** finito,
- d é uma função de Q × S em Q, denominada ***função de transição*** de M.

O autômato finito começa no estado q_0 e lê os caracteres de sua cadeia de entrada um por vez. Se o autômato está no estado q e lê o caractere de entrada a, passa ("faz uma transição") do estado q para o estado d (q, a). Sempre que seu estado atual q é um membro de A, a máquina M ***aceitou*** a cadeia lida até então. Uma entrada que não é aceita é ***rejeitada***.

Um autômato finito M induz uma função f denominada ***função estado final***, de S* a Q, tal que $f(w)$ é o estado em que M termina após ter escaneado a cadeia w. Assim, M aceita uma cadeia w se e somente se $f(w) \in A$. Definimos a função f recursivamente usando a função transição:

$$\phi(\varepsilon) = q_0,$$
$$\phi(wa) = \delta(\phi(w), a) \quad para\, w \in \Sigma^*, a \in \Sigma$$

Figura 32.6 Um autômato finito simples de dois estados com o conjunto de estados $Q = \{0, 1\}$, estado inicial $q_0 = 0$ e alfabeto de entrada S = {a,b}. (**a**) Uma representação tabular da função de transição d. (**b**) Um diagrama de transição de estados equivalente. O

estado 1, mostrado em negro, é o único estado aceitador Arestas dirigidas representam transições. Por exemplo, a aresta do estado 1 para o estado 0 identificada por b indica que $d(1, b) = 0$. Esse autômato aceita as cadeias que terminam em um número ímpar de "a" s. Mais precisamente, aceita uma cadeia x se e somente se $x = yz$, onde y = e ou y termina com um b e $z = a_k$, onde k é ímpar. Por exemplo, na entrada abaaa, incluindo o estado inicial, esse autômato segue a sequência de estados ⟨0, 1, 0, 1, 0, 1 ⟩, portanto aceita essa entrada. Para a entrada abbaa, ele segue a sequência de estados ⟨0, 1, 0, 0, 1, 0⟩ e, portanto, rejeita essa entrada.

Autômatos de correspondência de cadeias

Para um padrão P dado, construímos um autômato de correspondência de cadeias em uma etapa de pré-processamento antes de usá-lo para procurar a cadeia de texto. A Figura 32.7 ilustra o autômato para o padrão P = ababaca. Daqui em diante, suporemos que P seja uma cadeia de padrão fixo dada; por brevidade, não indicamos a dependência de P em nossa notação.

Para especificar o autômato de correspondência de cadeias relacionado a um padrão $P[1..m]$ dado, primeiro definimos uma função auxiliar s, denominada ***função sufixo*** correspondente a P. A função s mapeia S* para $\{0, 1, ..., m\}$ tal que $s(x)$ é o comprimento do prefixo mais longo de P que é um sufixo de x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}. \tag{32.3}$$

A função sufixo s é bem definida, visto que a cadeia vazia P_0 =   é um sufixo de toda cadeia. Como exemplos, para o padrão P = ab, temos s() = 0, s(ccaca) = 1 e s(ccab) = 2. Para um padrão P de comprimento m, temos s(x) = m se e somente se P x. Pela definição da função sufixo, x y implica s(x) ≤ s(x).

estado	entrada a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i —		1	2	3	4	5	6	7	8	9	10	11
$T[i]$ —		a	b	a	b	a	b	a	c	a	b	a
estado $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

Figura 32.7 (**a**) Um diagrama de transição de estados para o autômato de correspondência de cadeias que aceita todos as cadeias que terminam com a cadeia ababaca. O estado 0 é o estado inicial, e o estado 7 (mostrado em negro) é o único estado aceitador. Uma aresta dirigida do estado i para o estado j identificada por a representa $d(i, a) = j$. Exceto pela aresta do estado 7 para 02, as arestas dirigidas para a direita, que formam a "espinha dorsal" do autômato, representadas na figura pelas setas grossas em negro, correspondem a

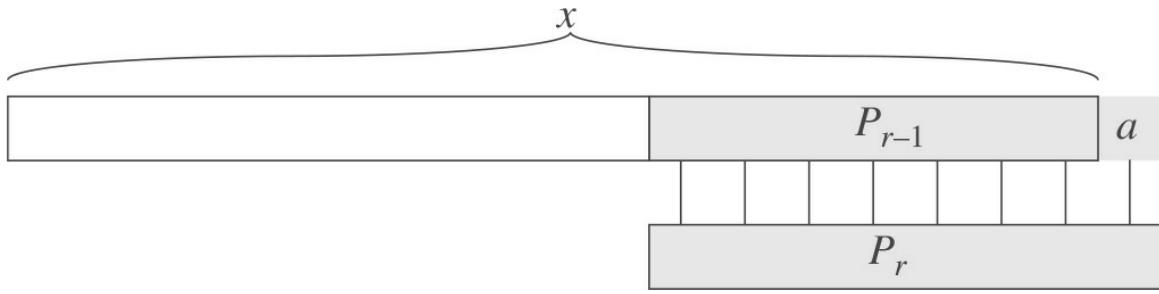
comparações bem-sucedidas entre caracteres do padrão e da entrada. Exceto pela aresta do estado 7 para 02, as arestas dirigidas para a esquerda correspondem a comparações mal-sucedidas. Algumas arestas correspondentes a comparações mal-sucedidas não são mostradas; por convenção, se um estado i não tem nenhuma aresta de saída identificada por a para algum $a \in$ S, então $d(i, a) = 0$. (**b**) A função transição d correspondente e a cadeia de padrão P = ababaca. As entradas correspondentes a comparações bem-sucedidas entre os caracteres do padrão e de entrada aparecem sombreadas. (**c**) A operação do autômato no texto T = abababacaba. Sob cada caractere de texto $T[i]$ é dado o estado $f(T_i)$ em que o autômato está depois de processar o prefixo T_i. O autômato encontra uma ocorrência do padrão e termina na posição 9.

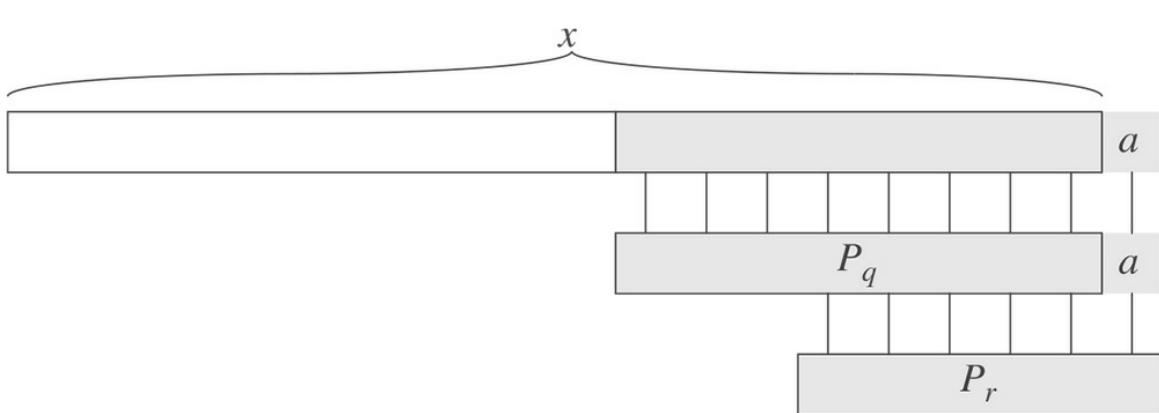
Definimos o autômato de correspondência de cadeias relativo a um padrão $P[1 .. m]$ dado da seguinte maneira:

- O conjunto de estados Q é $\{0, 1, ..., m\}$. O estado inicial q_0 é o estado 0, e o estado m é o único estado aceitador.
- A função transição d é definida pela seguinte equação, para qualquer estado q e caractere a:

$$\delta(q, a) = \sigma(P_q a) \tag{32.4}$$

Definimos $d(q, a) = s(P_q a)$ porque queremos manter o controle do prefixo mais longo do padrão P que correspondeu à corrente de texto T até agora. Consideramos os caracteres de T lidos mais recentemente. Para que uma subcadeia de T — digamos a subcorrente que termina em $T[i]$ — corresponda a algum prefixo P_j de P, esse prefixo P_j deve ser um sufixo de T_i. Suponha que $q = f(T_i)$, de modo que, após ler T_i, o autômato está no estado q. Projetamos a função transição d de modo que esse número de estado, q, nos informe o comprimento do prefixo mais longo de P que corresponde a um sufixo de T_i. Isto é, no estado q, P_q T_i e q (T_i). (Sempre que $q = m$, todos os m caracteres de P correspondem a um sufixo de T_i e, portanto, encontramos uma correspondência.) Assim, visto que $f(T_i)$ e $s(T_i)$ são iguais a q, veremos (no Teorema 32.4, mais adiante) que o autômato mantém o seguinte invariante:

$$\phi(T_i) = \sigma(T_i) \tag{32.5}$$

Se o autômato está no estado q e lê o próximo caractere $T[i + 1] = a$, então queremos que a transição leve ao estado correspondente ao prefixo mais longo de P que é um sufixo de $T_i a$, e esse estado é $s(T_i a)$. Como P_q é o prefixo mais longo de P que é um sufixo de T_i, o prefixo mais longo de P que é um sufixo de $T_i a$ não é somente $s(T_i a)$, mas também $s(P_q a)$. (O Lema 32.3 prova que $s(T_i a) = s(P_q a)$.) Assim, quando o autômato está no estado q, queremos que a função transição no caractere a leve o autômato para o estado $s(P_q a)$.

Há dois casos a considerar. No primeiro caso, $a = P[q + 1]$ de modo que o caractere a continua a corresponder ao padrão; nesse caso, como $d(q, a) = q + 1$, a transição continua a ocorrer ao longo da "espinha dorsal" do autômato (arestas grossas em negro na Figura 32.7). No segundo caso, $a \neq P[q + 1]$, de modo que a não continua a corresponder ao padrão. Aqui, temos de encontrar um prefixo menor de P que também é um sufixo de T_i. Como a etapa de pré-processamento compara o padrão com ele mesmo quando criamos o autômato de correspondência de cadeias, a função transição identifica rapidamente o mais longo de tais prefixos mais curtos de P.

Vamos examinar um exemplo. O autômato de correspondência de cadeias da Figura 32.7 tem $d(5, \text{c}) = 6$, ilustrando o primeiro caso, no qual a correspondência continua. Para ilustrar o segundo caso, observe que o autômato da Figura 32.7 tem $d(5, \text{b}) = 4$. Fazemos essa transição porque, se o autômato lê um b no estado $q = 5$, então $P_q\text{b}$ = ababab, e o prefixo mais longo de P que também é um sufixo de ababab é P_4 = abab.

Para esclarecer a operação de um autômato de correspondência de cadeias, damos agora um programa simples e eficiente para simular o comportamento de tal autômato (representado por sua função transição d) ao encontrar ocorrências de um padrão P de comprimento m em um texto de entrada $T[1 .. n]$. Como em qualquer autômato de correspondência de cadeias para um padrão de comprimento m, o conjunto de estado Q é $\{0, 1, ..., m\}$, o estado inicial é 0 e o único estado aceitador é o estado m.

Figura 32.8 (**a**) Ilustração para a prova do Lema 32.2. A figura mostra que $r \leq s(x) + 1$, onde $r = s(x\ a)$.

Figura 32.9 Ilustração para a prova do Lema 32.3. A figura mostra que $r = s(P_q\ a)$, onde $q = s(x)$ e $r = s(xa)$.

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n = T. comprimento
2  q = 0
3  for i = 1 to n
4       q = δ(q, T[i])
5       if q == m
6            imprimir "Padrão ocorre com deslocamento" i – m
```

Pela estrutura de laço simples de Finite-Automaton-Matcher é fácil ver que seu tempo de correspondência para uma cadeia de texto de comprimento n é Q(n). Porém, esse tempo de correspondência não inclui o tempo de pré-processamento necessário para calcular a função de transição d. Abordaremos esse problema mais adiante, depois de provar que o procedimento Finite-Automaton-Matcher funciona corretamente.

Considere como o autômato funciona em um texto de entrada $T[1\ ..\ n]$. Provaremos que o autômato está no estado $s(T_i)$ depois de examinar o caractere $T[i]$. Visto que $s(T_i) = m$ se e somente se P T_i, a máquina está no estado aceitador m se e somente se acabou de examinar P. Para provar esse resultado, fazemos uso dos dois lemas a seguir sobre a função sufixo s.

Lema 32.2 (Desigualdade da função sufixo)

Para qualquer cadeia x e caractere a, temos $s(xa) \leq s(x) + 1$.

Prova Referindo-nos à Figura 32.8, seja $r = s(xa)$. Se $r = 0$, então a conclusão $s(xa) = r \leq s(x) + 1$ é satisfeita trivialmente, pela não negatividade de $s(x)$. Agora, suponha que $r > 0$. Então, P_r xa, pela definição de s. Assim, P_{r-1}

x, descartando o a do final de P_r e do final de xa. Portanto, $r - 1 \leq s(x)$, já que $s(x)$ é o maior k tal que P_k x e $s(xa) = r \leq s(x) + 1$.

Lema 32.3 (Lema de recursão da função sufixo)

Para qualquer cadeia x e caractere a, se $q = s(x)$, então $s(xa) = s(P_q\, a)$.

Prova Pela definição de s, temos P_q x. Como mostra a Figura 32.9, temos também $P_q a$ xa. Se fizermos $r = s(xa)$, então P_r xa e, pelo Lema 32.2, $r \leq q + 1$. Assim, temos $|P_r| = r \leq q + 1 = |P_q a|$. Visto que $P_q a$ xa, P_r xa e $|P_q| \leq |P_q a|$, o Lema 32.1 implica que P_r $P_q a$. Portanto, $r \leq s\,(P_q a)$, isto é, $s(xa) \leq s(P_q a)$. Mas temos também $s(P_q a) \leq s(xa)$ visto que $P_q a$ xa. Assim, $s(xa) = s\,(P_q a)$.

Agora, estamos prontos para provar nosso teorema principal que caracteriza o comportamento de um autômato de correspondência de cadeias para um texto de entrada dado. Como observamos antes, esse teorema mostra que o autômato está simplesmente controlando, em cada etapa, o prefixo mais longo do padrão que é um sufixo do que foi lido até o momento. Em outras palavras, o autômato mantém o invariante (32.5).

Teorema 32.4

Se f é a função estado final de um autômato de correspondência de cadeias para um dado padrão P e $T[1 \,..\, n]$ é um texto de entrada para o autômato, então

$$\phi(T_i) = \sigma(T_i)$$

para $i = 0, 1, ..., n$.

Prova A prova é por indução em i. Para $i = 0$, o teorema é trivialmente verdadeiro, já que T_0 = E. Portanto, $f(T_0) = 0 = s(T_0)$.

Agora, supomos que $f(T_i) = s(T_i)$ e provamos que $f(T_i{+}1) = s(T_i{+}1)$. Representando $f(T_i)$ por q e $T[i + 1]$ por a, temos

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && \text{(pelas definições de } T_{i+1} \text{ e } a\text{)} \\
&= \delta(\phi(T_i), a) && \text{(pela definição de } \phi\text{)} \\
&= \delta(q, a) && \text{(pela definição de } q\text{)} \\
&= \sigma(P_q a) && \text{(pela definição (32.4) de } \delta\text{)} \\
&= \sigma(T_i a) && \text{(pelo Lema 32.3 e por indução)} \\
&= \sigma(T_{i+1}) && \text{(pela definição de } T_{i+1}\text{).}
\end{aligned}$$

Pelo Teorema 32.4, se a máquina entra no estado q na linha 4, então q é o maior valor tal que P_q T_i. Assim, temos $q = m$ na linha 5 se e somente se a máquina acabou de examinar uma ocorrência do padrão P. Concluímos que Finite-Automaton-Matcher funciona corretamente.

Cálculo da função transição

O procedimento a seguir calcula a função transição d por um padrão $P[1 \,..\, m]$ dado.

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m = P. comprimento
2  for q = 0 to m
3      for cada caractere a ∈ Σ
4          k = min(m + 1, q + 2)
5      repeat
6          k = k − 1
7      until P_k ⊐ P_q a
8      δ(q, a) = k
9  return δ
```

Esse procedimento calcula $d(q, a)$ de maneira direta, de acordo com sua definição na equação (32.4). Os laços aninhados que começam nas linhas 2 e 3 consideram todos os estados q e todos os caracteres a, e as linhas 4–8 definem $d(q, a)$ como o maior k tal que P_k $P_q a$. O código começa com o maior valor concebível de k, que é min(m, q + 1). Então, diminui k até P_k $P_q a$, o que a certa altura deve ocorrer, visto que P_0 = é um sufixo de toda cadeia.

O tempo de execução de Compute-Transition-Function é $O(m_3\ |S|)$ porque os laços exteriores contribuem com um fator de $m|S|$, o laço **repeat** interno pode ser executado no máximo $m + 1$ vezes, e o teste P_k $P_q a$ na linha 7 pode exigir a comparação de até m caracteres. Existem procedimentos muito mais rápidos; se usarmos algumas informações sobre o padrão P inteligentemente calculadas (veja o Exercício 32.4-8) podemos melhorar até $O(m\ |S|)$ o tempo requerido para calcular d a partir de P. Com esse procedimento melhorado para calcular d, podemos encontrar todas as ocorrências de um padrão de comprimento m em um texto de comprimento n que utilizou um alfabeto no tempo de pré-processamento $O(m\ |S|)$ e tempo de correspondência $Q(n)$.

Exercícios

32.3-1 Construa o autômato de correspondência de cadeias para o padrão P = aabab e ilustre como ele funciona na cadeia de texto T = aaababaabaababaab.

32.3-2 Desenhe um diagrama de transição de estados para um autômato de correspondência de cadeias para o padrão ababbabbababbababbabb no alfabeto S = {a,b}.

32.3-3 Dizemos que um padrão P é ***sem sobreposições*** se P_k P_q implica $k = 0$ ou $k = q$. Descreva o diagrama de transição de estados do autômato de correspondência de cadeias para um padrão sem sobreposições.

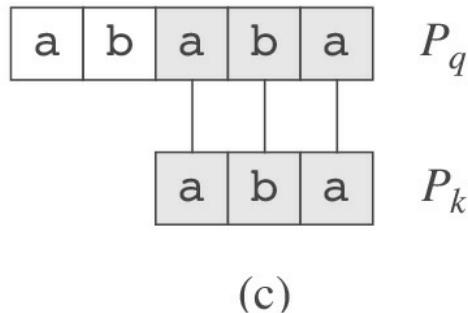
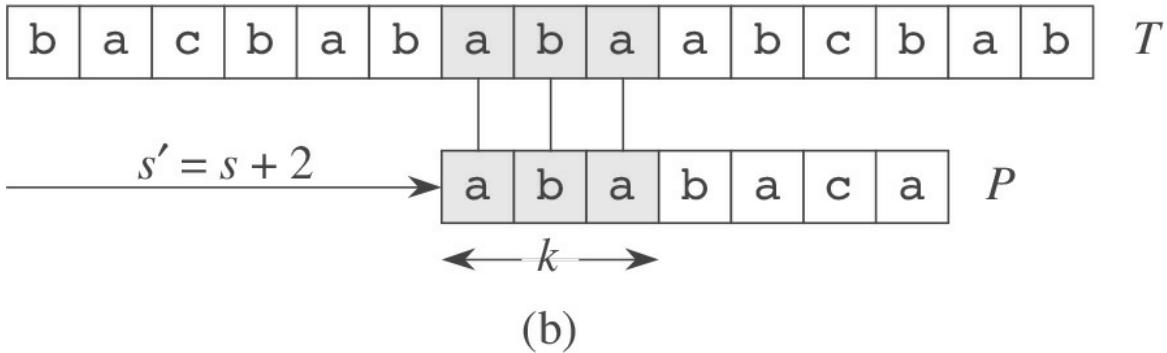
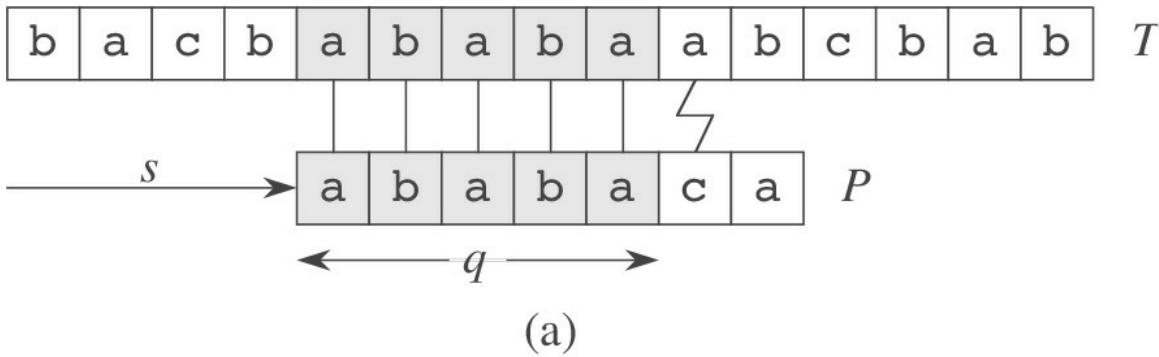
32.3-4 ★ Dados dois padrões P e P', descreva como construir um autômato finito que determina todas as ocorrências de quaisquer desses padrões. Procure minimizar o número de estados em seu autômato.

32.3-5 Dado um padrão P que contém caracteres lacuna (veja o Exercício 32.1-4), mostre como construir um autômato finito que possa encontrar uma ocorrência de P em um texto T no tempo de correspondência $O(n)$, onde $n = |T|$.

32.4 ★ O ALGORITMO KNUTH-MORRIS-PRATT

Agora, apresentamos um algoritmo de correspondência de cadeias de tempo linear criado por Knuth, Morris e Pratt. Esse algoritmo evita totalmente o cálculo da função transição d, e seu tempo de correspondência é Q(n) usando apenas uma função auxiliar p, que calculamos antecipadamente a partir do padrão no tempo Q(m) e armazenamos em um arranjo $p[1 \, ..m]$. O arranjo p nos permite calcular a função de transição d eficientemente (em um sentido amortizado) durante a execução conforme necessário. Em termos aproximados, para qualquer estado $q = 0, 1, ..., m$ e qualquer caractere $a \in$ S, o valor $p[q]$ contém as informações de que precisamos para calcular $d(q, a)$, mas que não dependem de a. Visto que o arranjo p tem apenas m entradas enquanto d tem Q(m |S|) entradas, economizamos um fator de |S| no tempo de pré-processamento calculando p em vez de d.

A função prefixo para um padrão

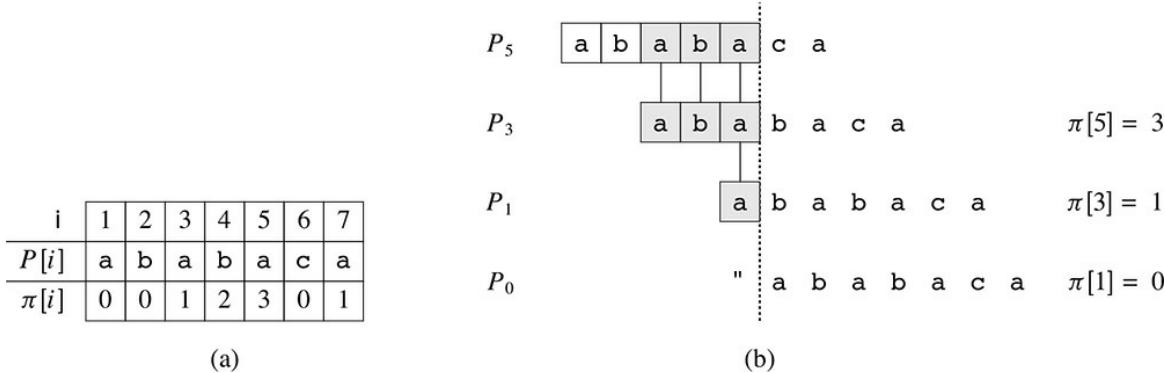
A função prefixo p para um padrão captura conhecimento sobre as corrrespondências entre o padrão e deslocamentos dele próprio. Podemos aproveitar essa informação para evitar testes de deslocamentos inúteis no algoritmo ingênuo de correspondência de padrões e para evitar o cálculo antecipado da função de transição completa d para um autômato de correspondência de cadeias.

Considere a operação do algoritmo ingênuo. A Figura 32.10(a) mostra um determinado deslocamento s de um gabarito que contém o padrão P = ababaca quando comparado com um texto T. Para esse exemplo, $q = 5$ dos caracteres corresponderam aos do texto, mas o sexto caractere não correspondeu ao caractere do texto. A informação de que q caracteres do padrão corresponderam aos do texto determina os caracteres de texto correspondentes. Saber quais são esses q caracteres de texto nos permite determinar imediatamente que certos deslocamentos são não válidos. No exemplo da figura, o deslocamento $s + 1$ é necessariamente não válido, já que o primeiro caractere do padrão, (a), estaria alinhado com um caractere de texto que sabemos que não corresponde ao primeiro caractere do padrão, mas corresponde ao segundo caractere do padrão, (b). Contudo, o deslocamento $s' = s + 2$ mostrado na parte (b) da figura alinha os três primeiros caracteres do padrão com três caracteres de texto que devem necessariamente ser correspondentes. Em geral, é útil saber a resposta para a seguinte pergunta:

Dado que caracteres do padrão $P[1 \, .. \, q]$ correspondem a caracteres de texto $T[s + 1 \, .. \, s + q]$, qual é o menor deslocamento $s' > s$ tal que, para algum $k < q$,

$$P[1 \, .. \, k] = T[s' + 1 \, .. \, s' + k] \, , \tag{32.6}$$

onde $s' + k = s + q$?

Em outras palavras, sabendo que P_q T_{s+q}, queremos o prefixo próprio mais longo P_k de P_q que é também um sufixo de T_{s+q}. (Visto que $s' + k = s + q$, se tivermos s e q, encontrar o menor deslocamento s' equivale a encontrar o maior comprimento de prefixo k.) Somamos a diferença $q - k$ nos comprimentos desses prefixos de P ao deslocamento s' para chegarmos ao nosso novo deslocamento s', de modo que $s' = s + (q - k)$. No melhor caso, $k = 0$, de modo que $s' = s + q$, e descartamos imediatamente os deslocamentos $s + 1$, $s + 2$, ..., $s + q - 1$. Em qualquer caso, no novo deslocamento s_o não precisamos comparar os k primeiros caracteres de P com os caracteres correspondentes de T, visto que a equação (32.6) garante que eles correspondem.

Figura 32.10 A função prefixo *p*. (**a**) O padrão *P* = ababaca se alinha com um texto *T* de modo que os primeiros *q* = 5 caracteres são correspondentes. Os caracteres correspondentes, sombreados, estão ligados por linhas verticais. (**b**) Usando somente o nosso conhecimento dos cinco caracteres correspondentes, podemos deduzir que um deslocamento de *s* + 1 é não válido, mas que um deslocamento de *s'* = *s* + 2 é compatível com tudo o que sabemos sobre o texto, e portanto é potencialmente válido. (**c**) Podemos calcular antecipadamente informações úteis para tais deduções comparando o padrão com ele próprio. Aqui, vemos que o prefixo mais longo de *P* que também é um sufixo de P_5 é P_3 . Representamos essa informação calculada antecipadamente no arranjo *p*, de modo que *p*[5] = 3. Dado que *q* caracteres são correspondentes no deslocamento *s*, o próximo deslocamento potencialmente válido é em *s'* = *s* + (*q* – *p* [*q*]), como mostra a parte (b).

Podemos calcular antecipadamente a informação comparando o padrão com si mesmo, como demonstra a Figura 32.10(c). Visto que $T[s' + 1 \mathinner{..} s' + k]$ é parte da porção conhecida do texto, é um sufixo da cadeia P_q. Portanto, podemos interpretar que a equação (32.6) solicita o maior $k < q$ tal que P_k P_q. Então, o novo deslocamento $s' = s + (q - k)$ é o próximo deslocamento potencialmente válido. Veremos que é conveniente armazenar, para cada valor de *q*, o número *k* de caracteres correspondentes no novo deslocamento *s*′, em vez de armazenar, digamos, $s' - s$.

Formalizamos a informação que pré-computamos da seguinte maneira. Dado um padrão $P[1..m]$, a ***função prefixo*** para o padrão *P* é a função $p : \{1, 2, ..., m\} \rightarrow \{0, 1, ..., m - 1\}$ tal que

$$\pi[q] = \max\{k : k < q \text{ e } P_k \sqsupset P_q\} .$$

Isto é, $p[q]$ é o comprimento do prefixo mais longo de P que é um sufixo próprio de P_q. A Figura 32.11(a) dá a função prefixo completa p para o padrão ababaca.

O pseudocódigo a seguir dá o algoritmo de correspondência Knuth-Morris-Pratt como o procedimento KMp-Matcher. Em sua maior parte, o procedimento decorre de Finite-Automaton-Matcher, como veremos. KMp-Matcher chama o procedimento auxiliar Compute-prefix-Function para calcular p.

Figura 32.11 Ilustração do Lema 32.5 para o padrão P ababaca e $q = 5$. (**a**) A função p para o padrão dado. Visto que $p[5] = 3$, $p[3] = 1$ e $p[1] = 0$, iterando p obtemos $p^*[5] = \{3,1, 0\}$. (**b**) Deslizamos o gabarito que contém o padrão P para a direita e observamos quando algum prefixo P_k de P corresponde a algum sufixo próprio de P_5 ; conseguimos correspondências quando $k = 3,1$ e 0. Na figura, a primeira linha dá P, e a linha vertical pontilhada é desenhada logo após P_5 . Linhas sucessivas mostram todos os deslocamentos de P que resultam na concordância entre algum prefixo P_k de P e algum sufixo de P_5 . Caracteres sucessivamente correspondentes são sombreados. Linhas verticais ligam caracteres correspondentes alinhados. Assim, $\{k : k < 5 \text{ e } P_k \;\; P_5\} = \{3, 1, 0\}$. O Lema 32.5 afirma que $p^*[q] = \{k : k < q \text{ e } P_k \;\; P_q\}$ para todo q.

```
KMP-MATCHER(T, P)
 1  n = T.comprimento
 2  m = P.comprimento
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                   // número de caracteres correspondentes
 5  for i = 1 to n                          // varre o texto da esquerda para a direita
 6       while q > 0 e P[q + 1] ≠ T[i]
 7            q = π[q]                      // próximo caractere não é correspondente
 8       if P[q + 1] == T[i]
 9            q = q + 1                     // próximo caractere é correspondente
10       if q == m                          // P inteiro é correspondente?
11       imprimir "Padrão ocorre com deslocamento" i – m
12       q = π[q]                            // procurar próxima correspondência
```

```
COMPUTE-PREFIX-FUNCTION. (P)
 1  m = P. comprimento
 2  seja π [1. . m] um novo arranjo
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6       while k > 0 e P[k + 1] ≠ P[q]
 7            k = π[k]
 8       if P[k + 1]  == P[q]
 9            k = k + 1
10       π[q] = k
11  return π
```

Esses dois procedimentos têm muito em comum porque ambos comparam uma cadeia com o padrão P: Kmp-Matcher compara o texto T com P, e Compute-prefix-Function compara P com ele mesmo.

Começamos com uma análise dos tempos de execução desses procedimentos. Provar que os procedimentos são corretos será mais complicado.

Análise do tempo de execução

O tempo de execução de Compute-prefix-Function é Q(m), o que mostramos usando o método agregado de análise amortizada (veja a Seção 17.1). A única parte complicada é mostrar que o laço **while** das linhas 6–7 é executado $O(m)$ vezes no total. Mostraremos que ele faz no máximo $m - 1$ iterações. Começamos com algumas observações sobre k. A primeira é que a linha 4 inicia k em 0, e a única maneira de k aumentar é pela operação de incremento na linha 9, que é executada no máximo uma vez por iteração do laço **for** das linhas 5–10. Assim, o aumento total total em k é no máximo $m - 1$. A segunda é que, visto que $k < q$ quando entra no laço **for** e cada iteração do laço incrementa q, temos sempre $k < q$. Portanto, as atribuições nas linhas 3 e 10 garantem que $p[q] < q$ para todo $q = 1, 2,..., m$, o que significa que cada iteração do laço **while** diminui k. A terceira é que k nunca se torna negativo. Juntando esses fatos, vemos que a redução total em k resultante do laço **while** é limitada por cima pelo aumento total de k em todas as iterações do laço **for**, que é $m - 1$. Assim, o laço **while** itera no máximo $m - 1$ vezes no todo, e Compute-prefixfunction é executado no tempo Q(m).

O Exercício 32.4-4 pede que você mostre, por uma análise agregada semelhante, que o tempo de correspondência de KMp-Matcher é Q(n).

Comparado com Finite-Automaton-Matcher, usando p em vez de d, reduzimos o tempo de pré-processamento do padrão de $O(m\ |S|)$ para $Q(m)$, mantendo o tempo real de correspondência limitado por $Q(n)$.

Correção do cálculo da função prefixo

Veremos, um pouco mais adiante, que a função prefixo p nos ajuda a simular a função transição d em um autômato de correspondência de cadeias. Porém, em primeiro lugar precisamos provar que o procedimento Compute-prefix-Function realmente calcula a função prefixo corretamente. Para tal, precisaremos encontrar todos os prefixos P_k que são sufixos próprios de um prefixo P_q. dado. O valor de $p[q]$ nos dá tal prefixo mais longo, mas o lema apresentado a seguir, ilustrado na Figura 32.11, mostra que iterando a função prefixo p, podemos de fato enumerar todos os prefixos P_k que são sufixos próprios de P_q. Seja

$$\pi^*[q] = \pi[q] =, \pi^{(2)}[q] =, \pi^{(3)}[q], ..., \pi^{(t)}[q]\} \ ,$$

onde $p^{(i)}[q]$ é definida em termos de iteração funcional, de modo que $p^{(0)}[q] = q$ e $p^{(i)}[q] = p^{(i-1)}[q]]$ para $i \geq 1$ e onde a sequência em $p^*[q]$ para quando $p^{(t)}[q] = 0$.

Lema 32.5 (Lema da iteração da função prefixo)

Seja P um padrão de comprimento m com função prefixo p. Então, para $q = 1, 2, ...,m$, temos $p^*[q] = \{k : k < q$ e P_k $P_q\}$.

Prova Primeiro provamos que $p[q] \subseteq \{k : k < q$ e P_k $P_q\}$ ou, o que é equivalente,

$$i \in \pi^*[q] \text{ implica } P_i \sqsupset P_q. \qquad (32.7)$$

Se $i \in p^*[q]$, então $i = p^{(u)}[q]$ para algum $u > 0$. Provamos a equação (32.7) por indução em u. Para $u = 1$, temos $i = p[q]$, e a afirmação decorre, já que $i < q$ e $P_{p[q]}$ P_q pela definição de p. Usando as relações $p[i] < i$ e $P_{p[i]}$ P_i e a transitividade de < e estabelece a afirmação para todo i em $p^*[q]$. Portanto, $p^*[q] \subseteq \{k : k < q$ e P_k $P\}$.

Agora provamos que $\{k : k < q$ e P_k $P_q\} \subseteq p^*[q]$ por contradição. Suponha, ao contrário, que o conjunto $\{k : k < q$ e P_k $P_q\} - p^*[q]$ é não vazio e seja j o maior número no conjunto. Como $p[q]$ é o maior valor em $\{k : k < q$ e P_k $P_q\}$ e $p[q] \in p^*[q]$, devemos ter $j < p[q]$ e, assim, denotamos por j' o menor inteiro em $p^*[q]$ que é maior que j. (Podemos escolher $j = p[q]$, se nenhum outro número em $p^*[q]$ é maior que j.) Temos P_j P_q porque $j \in \{k : k < q$ e P_k $P_q\}$ e por j' e $p^*[q]$ e a equação (32.7) temos $P_{j'}$, P_q . Assim, P_j $P_{j'}$; pelo Lema 32.1 e j é o maior valor menor que j' com essa propriedade. Portanto, devemos ter $p[j'] = j$ e, visto que $j' \in p^*[q\}$, devemos ter também $j \in p^*[q\}$. Essa contradição prova o lema.

O algoritmo Compute-prefix-Function calcula $p[q]$, em ordem, para $q = 1, 2, ..., m$. Fazer $p[1] = 0$ na linha 3 de Compute-prefix-Function certamente é correto, já que $p[q] < q$ para todo q. Usaremos o lema apresentado a seguir e seu corolário para provar que Compute-prefix-Function calcula $p[q]$ corretamente para $q > 1$.

Lema 32.6

Seja P um padrão de comprimento m e seja p a função prefixo para P. Para $q = 1, 2, ..., m$, se $p[q] > 0$, então $p[q] - 1 \in p^*[q-1]$.

Prova Se $r = p[q] > 0$, então $r < q$ e P_r P_q; portanto, $r - 1 < q - 1$ e P_{r-1} P_{q-1} (descartando o último caractere de P_r e P_q), o que podemos fazer porque $r > 0$). Portanto, pelo Lema 32.5, $r - 1 \in r^*[q-1]$. Assim, temos $p[q] - 1 = r - 1 \in p^*[q-1]$.

Para $q = 2, 3, ..., m$, defina o subconjunto $E_q - 1 \subseteq p^*[q-1]$ por

$$\begin{aligned} E_q - 1 &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} \\ &= \{k : k < q-1 \text{ e } P_k \sqsupset P_{q-1} \text{ e } P[k+1] = P[q]\} \text{ (pelo Lema 32.5)} \\ &= \{k : k < q-1 \text{ e } P_{k+1} \sqsupset P_q\}. \end{aligned}$$

O conjunto E_{q-1} consiste nos valores $k < q - 1$ para os quais $P_k \quad P_{q-1}$ e para os quais, como $P[k + 1] = P[q]$, temos $P_{k+1} \quad P_q$. Assim, E_{q-1} consiste nesses valores $k \in p^*[q - 1]$ tais que podemos estender P_k a P_{k+1} e obter um sufixo próprio de P_q.

Corolário 32.7

Seja P um padrão de comprimento m e seja p a função prefixo para P. Para $q = 2, 3, ..., m$,

$$\pi[q] = \begin{cases} 0 & se\ E_{q-1} = \varnothing \\ 1 + \max\{k \in E_{q-1}\} & se\ E_{q-1} \neq \varnothing \end{cases}$$

Prova Se E_{q-1} é vazio, não existe nenhum $k \in p^*[q - 1]$ (incluindo $k = 0$) para o qual possamos estender P_k a P_{k+1} e obter um sufixo próprio de P_q. Portanto, $p[q] = 0$.

Se E_{q-1} é não vazio, então para cada $k \in E_{q-1}$ temos $k + 1 < q$ e $P_{k+1} \quad P_q$. Portanto, pela definição de $p[q]$, temos

$$\pi[q] \geq 1 + \max\{k \in E_{q-1}\}. \tag{32.8}$$

Observe que $p[q] > 0$. Seja $r = p[q] - 1$, de modo que $r + 1 = p[q]$ e, portanto, $P_{r+1} \quad P_q$. Visto que $r + 1 > 0$, temos $P[r + 1 = \quad P[q]$. Além disso, pelo Lema 32.6, temos $r \in p^*[q - 1]$. Portanto, $r \in E_{q-1}$ e, assim, $r \leq \max\{k \in E_{q-1}\}$ ou, o que é equivalente,

$$\pi[q] \leq 1 + \max\{k \in E_{q-1}\}. \tag{32.9}$$

Combinando as equações (32.8) e (32.9), conclui-se a prova.

Agora, terminamos a prova de que Compute-prefix-Function calcula p corretamente. No procedimento Compute-prefix-Function, no início de cada iteração do laço **for** das linhas 5–10, temos que $k = p[q - 1]$. Essa condição é imposta pelas linhas 3 e 4 quando entramos no laço pela primeira vez e permanece verdadeira em cada iteração sucessiva, por causa da linha 10. As linhas 6–9 ajustam k de modo que ele se torna o valor correto de $p[q]$. O laço **while** nas linhas 6–7 pesquisa todos os valores $k \in p^*[q - 1]$ até encontrar um para o qual $P[k + 1] = P[q]$; nesse ponto, k é o maior valor no conjunto E_{q-1}, de modo que, pelo Corolário 32.7, podemos definir $p[q]$ como $k + 1$. Se o laço **while** não puder encontrar um $k \in p^*[q - 1]$ tal que $P[k + 1] = P[q]$, então k é igual a 0 na linha 8. Se $P[1] = P[q]$, então devemos definir k e $p[q]$ como 1; caso contrário, devemos deixar k como está e definir $p[q]$ como 0. As linhas 8–10 definem k e $p[q]$ corretamente em qualquer caso. Isso conclui nossa prova da correção de Compute-prefix-Function.

Correção do algoritmo Knuth-Morris-Pratt

Podemos considerar o procedimento KMp-Matcher como uma versão reimplementada do procedimento Finite-Automaton-Matcher, mas usando a função prefixo p para calcular transições de estado. Especificamente, provaremos

que na i-ésima iteração dos laços for de ambos, KMp-Matcher e Finite-Automaton-Matcher, o estado q tem o mesmo valor quando testamos a igualdade com m (na linha 10 em KMp-matcher e na linha 5 em Finite- Automaton-Matcher). Uma vez demonstrado que KMp-Matchersimula o comportamento de Finite-Automaton-matcher, a correção de KMp-Matcher decorre da correção de Finite-Automaton-Matcher (embora vejamos mais adiante por que a linha 12 em KMp-Matcher é necessária).

Antes de provarmos formalmente que Kmp-matcher simula corretamente Finite-Automaton-Matcher, vamos gastar um instante para entender como a função prefixo p substitui a função transição d. Lembre-se de que, quando um autômato de correspondência de cadeias está no estado q e varre um caractere $a = T[i]$, ele passa para um novo estado $d(q, a)$. If a $= P[q + 1]$, de modo que a continua a corresponder ao padrão, então $d(q, a) = q + 1$. Caso contrário, a $\neq P[q + 1\}$, de modo que a não continua a corresponder ao padrão, e $0 \leq d(q, a) \leq q$. No primeiro caso, quando a continua a corresponder, KMp-Matcher passa para o estado $q + 1$ sem referenciar a função p: o laço **while** na linha 6 resulta falso na primeira vez, o teste na 8 resulta verdadeiro e a linha 9 incrementa q.

A função p entra em ação quando o caractere a não continua a corresponder ao padrão, de modo que o novo estado $d(q, a)$ é q ou está à esquerda de q ao longo da espinha dorsal do autômato. O laço **while** das linhas 6–7 em KMp-Matcher itera pelos estados em $p^*[q]$, e para ou quando chega a um estado, digamos q', tal que a corresponde a $P[q' + 1]$ ou q' já tenha percorrida todo o caminho descendente até 0. Se a corresponde a $P[q' + 1]$, então a linha 9 define o novo estado como $q' + 1$, que deve ser igual a $d(q, a)$ para que a simulação funcione corretamente. Em outras palavras, o novo estado deve ser ou o estado 0 ou um a mais do que algum estado em $p^*[q]$.

Vamos examinar as Figuras 32.7 e 32.11, onde os exemplos são para o padrão P = ababaca. Suponha que o autômato esteja no estado $q = 5$; os estados em $p^*[5]$ são, em ordem descendente, 3, 1 e 0. Se o próximo caractere verificado é c, então é fácil ver que o autômato passa para o estado $d(5, \text{c}) = 6$ em Finite-Automaton-Matcher e também em KMp-Matcher. Agora suponha que o próximo caractere verificado seja b, de modo que o autômato deve passar para o estado $d(5, \text{b}) = 4$. O laço **while** em KMp-Matcher sai após executar a linha 7 uma vez e chega ao estado $q' = p[5] = 3$. Visto que $P[q' + 1] = P(4) = \text{b}$, o teste na linha 8 revela-se verdadeiro e KMp-Matcher passa para novo estado $q' + 1 = 4 = d(5, \text{b})$. Finalmente, suponha que o próximo caractere escaneado seja a, de modo que o autômato deve passar para o estado $d(5, \text{a}) = 1$. Nas três primeiras vezes em que o teste na linha 6 é executado ele dá verdadeiro. Na primeira vez, verificamos que $P[6] = \text{c} \neq \text{a}$, e Kmp-Matcher passa para o estado $p[5] = 3$ (o primeiro estado em $p^*[5]$). Na segunda vez, verificamos que $P[4] = \text{b} \neq \text{a}$ e passamos para o estado $p[3] = 1$ (o segundo estado em $p^*[5]$). Na terceira vez, verificamos que $\text{P}[2] = \text{b} \neq \text{a}$ e passamos para o estado $p[1] = 0$ (o último estado em $p^*[5]$). O laço **while** sai uma vez e chega ao estado $q' = 0$. Agora, a linha 8 descobre que $P[q' + 1] = P[1] = \text{a}$, e a linha 9 passa o autômato para o novo estado $q' + 1 = 1 = d\ (5, \text{a})$.

Assim, nossa intuição é que Kmp-Matcher itera em todos os estados em $p^*[q]$ em ordem decrescente, parando em algum q' e então possivelmente passando para o estado $q' + 1$. Embora isso possa parecer muito trabalho só para simular o cálculo de $d[q, a]$, não esqueça que assintoticamente KMp-Matcher não é mais lento que Finite-Automaton-Matcher.

Agora estamos prontos para provar formalmente a correção do algoritmo Knuth-Morris-Pratt. Pelo Teorema 32.4, temos que $q = \text{s}(T_{\text{i}})$ após cada vez que executarmos a linha 4 de Finite-Automaton-Matcher. Portanto, basta mostrar que a mesma propriedade é válida em relação ao laço **for** em KMP-MATCHER. A prova é realizada por indução em relação ao número de iterações do laço. Inicialmente, ambos os procedimentos definem q como 0 quando entram em seus respectivos laços **for** pela primeira vez. Considere a iteração i do laço for laço em Kmp-matcher, e seja q' o estado no início dessa iteração do laço. Pela hipótese indutiva, temos $q' = \text{s}(T_{\text{i}-1})$. Precisamos mostrar que $q' = \text{s}(T_{\text{i}})$ na linha 10. (Novamente, trataremos a linha 12 separadamente.)

Quando consideramos o caractere $T[i]$, o prefixo mais longo de P que é um sufixo de T_{i} é $P_{\text{q}'} + 1$ (se $P[q' + 1] = \text{T}\ [i]$) ou algum prefixo (não necessariamente próprio, e possivelmente vazio) de $P_{\text{q}'}$. Consideramos separadamente os três casos nos quais $\text{s}(T_{\text{i}}) = 0$, $\text{s}(T_{\text{i}}) = q' + 1$ and $0 < \text{s}\ (T_{\text{i}}) \leq q'$.

- Se $\text{s}\ (T_{\text{i}}) = 0$, então $\text{P}_0 =$ é o único prefixo de P que é um sufixo de T_i. O laço **while** das linhas 6–7 itera pelos valores em $p^*[q']$, mas, embora P_q T_{i-1} para todo $q \in p^*[q']$, o laço nunca encontra um q tal que $P[q + 1] =$

$T[i]$. O laço termina quando q chega a 0 e, é claro, a linha 9 não é executada. Portanto, $q = 0$ na linha 10, de modo que $q = s(T_i)$.

- Se s $(T_i) = q' + 1$, então $P[q' + 1] = T[i]$, e o teste do laço **while** na linha 6 falha inteiramente na primeira vez. A linha 9 é executada, incrementando q de modo que, depois, temos $q = q' + 1 = s(T_i)$.
- Se $0 <$ s $(T_i) \leq q'$, então o laço **while** das linhas 6–7 itera no mínimo uma vez, verificando em ordem decrescente cada valor $q \in p^*[q']$ até parar em algum $q < q'$.

Assim, P_q é o prefixo mais longo de $P_{q'}$ para o qual $P[q + 1] = T[i]$, de modo que, quando o laço **while** termina, $q + 1 = s(P_q, T[i]))$. Visto que $q' = s(T_{i-1})$, o Lema 32.3 implica que $s(T_{i-1}T[i]) = s(P_{q'} T[i])$ Assim, temos

$$
\begin{aligned}
q+1 &= \sigma(P_{q,} T[i]) \\
&= \sigma(T_{i-1} T[i]) \\
&= \sigma(T_i)
\end{aligned}
$$

quando o laço **while** termina. Após a linha 9 incrementar q, temos $q = s(T_i)$.

A linha 12 é necessária em KMp-Matcher porque, caso contrário, poderíamos referenciar P[m + 1] na linha 6 após encontrar uma ocorrência de P. (O argumento de que $q = s(T_{i-1})$ na próxima execução da linha 6 permanece válido pela sugestão dada pelo Exercício 32.4-8: $d(m, a) = d(r,[m], a)$ ou, o que é equivalente, $s(Pa) = s\ (P_{p[m]}a)$ para qualquer $a \in \sum$.) O argumento restante para a correção do algoritmo de Knuth-Morris-Pratt decorre da correção deFinite-Automaton-Matcher, já que mostramos que KMp-Matcher simula o comportamento de Finite- automaton-Matcher.

Exercícios

32.4-1 Calcule a função prefixo p para o padrão ababbabbabbababbabb.

32.4-2 Dê um limite superior para o tamanho de $p^*[q]$ em função de q. Dê um exemplo para mostrar que seu limite é justo.

32.4-3 Explique como determinar as ocorrências do padrão P no texto T examinando a função p em busca da cadeia PT (a cadeia de comprimento $m + n$ que é a concatenação de P e T).

32.4-4 Use uma análise agregada para mostrar que o tempo de execução de Kmp-matcher é Q(n).

32.4-5 Use uma função de potencial para mostrar que o tempo de execução de KMp-Ma- tcher é Q(n).

32.4-6 Mostre como melhorar KMp-Matcher substituindo a ocorrência de p na linha 7 (mas não na linha 12) por p', onde p' é definido recursivamente para q = 1, 2, ..., m pela equação

$$
\pi'[q] = \begin{cases} 0 & se\ \pi[q] = 0 \\ \pi'[\pi[q]] & se\ \pi[q] \neq 0\ e\ P[\pi[q]+1] = P[q+1] \\ \pi[q] & se\ \pi[q] \neq 0\ e\ P[\pi[q]+1] \neq P[q+1] \end{cases}
$$

Explique por que o algoritmo modificado é correto e em que sentido essa modificação constitui uma otimização.

32.4-7 Dê um algoritmo de tempo linear para determinar se um texto T é uma rotação cíclica de outra cadeia T'. Por exemplo, as cadeias arc e car são rotações cíclicas uma da outra.

32.4-8 ★ Dê um algoritmo de tempo $O(m|\Sigma|)$ para calcular a função transição d para o autômato de correspondência de cadeias relativo a um padrão P dado. (*Sugestão:* Prove que $\delta(q, a) = \delta(\pi[q], a)$ se $q = m$ ou $P[q + 1] \neq a$.)

Problemas

32-1 ***Correspondência de cadeias baseada em fatores de repetição***

Seja y_i a concatenação da cadeia y com ela própria i vezes. Por exemplo, $(ab)^3$ = ababab. Dizemos que uma cadeia $x \in S^*$ tem ***fator de repetição*** r se $x = y_r$ para alguma cadeia $y \in S^*$ e algum $r > 0$. Seja (x) o maior r tal que x tenha fator de repetição r.

a. Dê um algoritmo eficiente que tome como entrada um padrão $P[1 .. m]$ e calcule o valor (P_i) para $i = 1, 2, ..., m$. Qual é o tempo de execução de seu algoritmo?

b. Para qualquer padrão $P[1 .. m]$, seja $*(P)$ definido como $\max_{1 \leq i \leq m}(P_i)$. Prove que, se o padrão P for escolhido aleatoriamente do conjunto de todos as cadeias binárias de comprimento m, então o valor esperado de $*(P)$ é $O(1)$.

c. Demonstre que o algoritmo de correspondência de cadeias a seguir encontra corretamente todas as ocorrências do padrão P em um texto $T[1 .. n]$ no tempo $O(*(P)n + m)$.

```
REPETITION-MATCHER(P, T)
1  m = P.comprimento
2  n = T.comprimento
3  k = 1 + ρ*(P)
4  q = 0
5  s = 0
6  while s ≤ n – m
7      if T[s + q + 1] = P[q + 1]
8           q = q + 1
9               if q == m
10                   imprimir "Padrão ocorre com deslocamento" s
11         if q == m ou T[s + q + 1] ≠ P[q + 1]
12              s = s + max(1, ⌈q/k⌉)
13                  q = 0
```

Esse algoritmo foi desenvolvido por Galil e Seiferas. Estendendo bastante essas ideias, eles obtêm um algoritmo de correspondência de cadeias de tempo linear que utiliza somente o espaço de armazenamento $O(1)$ além do que é necessário para P e T.

NOTAS DO CAPÍTULO

A relação entre correspondência de cadeias e a teoria de autômatos finitos é discutida por Aho, Hopcroft e Ullman [5]. O algoritmo Knuth-Morris-Pratt [214] foi criado independentemente por Knuth e Pratt e por Morris; eles publicaram seu trabalho em conjunto. Reingold, Urban e Gries [294] apresentam um tratamento alternativo para o algoritmo de Knuth-Morris-Pratt. O algoritmo Rabin-Karp foi proposto por Karp e Rabin [201]. Galil e Seiferas [126] apresentam um interessante algoritmo determinístico de tempo linear para correspondência de cadeias que utiliza somente o espaço $O(1)$ além do que é exigido para armazenar o padrão e o texto.

---

[1] Escrevemos Q($n - m + 1$) em vez de Q($n - m$) porque s adota $n - m + 1$ valores diferentes. O "+1" é significativo em um sentido assintótico porque, quando $m = n$, calcular o valor isolado t_s demora tempo Q(1), e não tempo Q(0).

33 GEOMETRIA COMPUTACIONAL

Geometria computacional é o ramo da ciência da computação que estuda algoritmos para resolver problemas geométricos. Na engenharia e na matemática modernas, a geometria computacional tem aplicações em áreas tão diversas quanto gráficos por computador, robótica, projeto com VLSI, projeto com o auxílio do computador, modelagem molecular, metalurgia, manufatura, padrões têxteis, silvicultura e estatística. A entrada para um problema de geometria computacional normalmente é uma descrição de um conjunto de objetos geométricos, como um conjunto de pontos, um conjunto de segmentos de reta ou os vértices de um polígono em ordem anti-horária. A saída, muitas vezes, é uma resposta a uma consulta sobre os objetos como, por exemplo, se quaisquer das retas se intercepta ou, talvez, um novo objeto geométrico, como a envoltória convexa (o menor polígono convexo envolvente) do conjunto de pontos.

Neste capítulo, examinamos alguns algoritmos de geometria computacional em duas dimensões, isto é, no plano. Representamos cada objeto de entrada por um conjunto de pontos $\{p_1, p_2, p_3, ...\}$, onde cada $p_i = (x_i, y_i)$ e $x_i, y_i \in$. Por exemplo, representamos um polígono P de n vértices por uma sequência $\langle p_0, p_1, p_2, ..., p_{n-1} \rangle$ de seus vértices na ordem em que aparecem no contorno de P. A geometria computacional também pode ser aplicada a três dimensões e até a espaços com um número mais alto de dimensões, mas tais problemas e suas soluções podem ser muito difíceis de visualizar. Entretanto, ainda que em duas dimensões, podemos ver uma boa amostra de técnicas de geometria computacional.

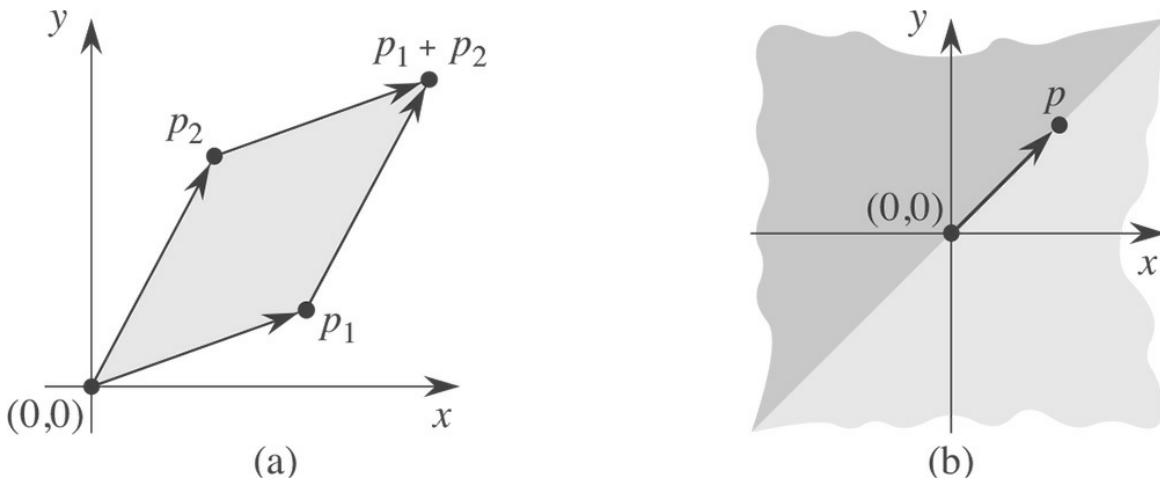
A Seção 33.1 mostra como responder eficientemente e com precisão a perguntas básicas sobre segmentos de reta: se um segmento está em sentido horário ou em sentido anti-horário em relação a um outro com o qual compartilha uma extremidade, para que lado viramos quando percorremos dois segmentos de reta adjacentes e se dois segmentos de reta se interceptam. A Seção 33.2 apresenta uma técnica denominada "varredura" que usamos para desenvolver um algoritmo de tempo $O(n \lg n)$ para determinar se um conjunto de n segmentos de reta contém alguma interseção. A Seção 33.3 apresenta dois algoritmos de "varredura rotacional" que calculam a envoltória convexa (o menor polígono convexo envolvente) de um conjunto de n pontos: a varredura de Graham, que é executada no tempo $O(n \lg n)$, e a marcha de Jarvis, que demora o tempo $O(nh)$, onde h é o número de vértices da envoltória convexa. Finalmente, a Seção 33.4 dá um algoritmo de divisão e conquista de tempo $O(n \lg n)$ para encontrar o par de pontos mais próximos em um conjunto de n pontos no plano.

33.1 PROPRIEDADES DE SEGMENTOS DE RETA

Vários algoritmos da geometria computacional neste capítulo exigem respostas a perguntas sobre as propriedades dos segmentos de reta. Uma ***combinação convexa*** de dois pontos distintos $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ é qualquer ponto $p_3 = (x_3, y_3)$ tal que, para algum α na faixa $0 \leq \alpha \leq 1$, temos $x_3 = \alpha x_1 + (1 - \alpha)x_2$ e $y_3 = \alpha y_1 + (1 - \alpha)y_2$. Escrevemos também que $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Intuitivamente, p_3 é qualquer ponto que está sobre a reta que passa por p_1 e p_2 e que está sobre ou entre p_1 e p_2 na reta. Dados dois pontos distintos p_1 e p_2, o ***segmento de reta*** $p_1 p_2$ é o conjunto de combinações convexas de p_1 e p_2. Denominamos p_1 e p_2 ***extremidades*** do segmento $p_1 p_2$. Às vezes, a

ordem de p_1 e p_2 é importante, e falamos do ***segmento dirigido*** $p_1 p_2$. Se p_1 é a ***origem*** (0, 0), então podemos tratar o segmento dirigido $p_1 p_2$ como o ***vetor*** p_2.

Nesta seção, exploraremos as seguintes questões:

1. Dados dois segmentos dirigidos $p_0 p_1$ e $p_0 p_2$, $p_0 p_1$ está em sentido horário em relação a $p_0 p_2$ considerando sua extremidade comum p_0?
2. Dados dois segmentos de reta $p_0 p_1$ e $p_1 p_2$, se percorrermos $p_0 p_1$ e depois $p_1 p_2$, fazemos uma curva para a esquerda no ponto p_1?
3. Os segmentos de reta $p_1 p_2$ e $p_3 p_4$ se interceptam? Não há nenhuma restrição nos pontos dados.

Podemos responder a cada pergunta no tempo $O(1)$, o que não deve ser nenhuma surpresa, já que o tamanho da entrada de cada pergunta é $O(1)$. Além disso, nossos métodos usarão apenas adições, subtrações, multiplicações e comparações. Não precisamos nem da divisão nem de funções trigonométricas, já que ambas podem ser dispendiosas em termos computacionais e propensas a problemas com erros de arredondamento. Por exemplo, o método “direto” de determinar se dois segmentos se interceptam — calcular a equação da reta da forma $y = mx + b$ para cada segmento (m é a inclinação e b é a interseção com o eixo y), determinar o ponto de interseção das retas e verificar se esse ponto está em ambos os segmentos — utiliza divisão para determinar o ponto de interseção. Quando os segmentos são quase paralelos, esse método é muito sensível à precisão da operação de divisão em computadores reais. O método desta seção, que evita divisão, é muito mais preciso.

Produtos cruzados

Calcular produtos cruzados está no núcleo de nossos métodos de segmentos de reta. Considere os vetores p_1 e p_2 mostrados na Figura 33.1(a). Podemos interpretar o ***produto cruzado*** $p_1 \times p_2$ como a área assinalada do paralelogramo formado pelos pontos (0, 0), p_1, p_2 e $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. Uma definição equivalente, porém mais útil, dá o produto cruzado como o determinante de uma matriz:[1]

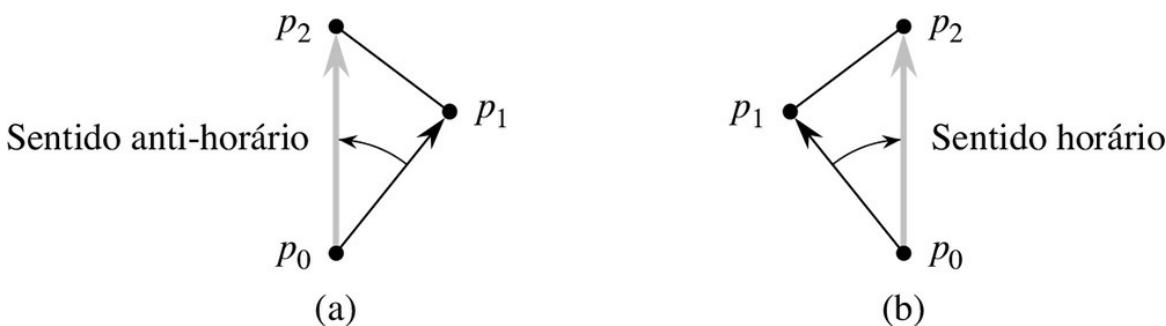
$$
\begin{aligned}
p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\
&= x_1 y_2 - x_2 y_1 \\
&= -p_2 \times p_1
\end{aligned}
$$

Figura 33.1 (**a**) O produto cruzado de vetores p_1 e p_2 é a área assinalada do paralelogramo. (**b**) A região sombreada em tom mais claro contém vetores que estão em sentido horário em relação a p. A região sombreada em tom mais escuro contém vetores que estão em sentido anti-horário em relação a p.

Se $p_1 \times p_2$ é positivo, então p_1 está em sentido horário em relação a p_2 em relação à origem (0, 0); se esse produto cruzado é negativo, então p_1 está em sentido anti-horário em relação a p_2 (veja o Exercício 33.1-1). A Figura 33.1(b) mostra as regiões horária e anti-horária em relação a um vetor p. Uma condição de contorno surge se o produto cruzado é zero; nesse caso, os vetores são ***colineares*** e apontam na mesma direção ou em direções opostas.

Para determinar se um segmento dirigido $\overrightarrow{p_0 p_1}$ está mais próximo de um segmento dirigido $\overrightarrow{p_0 p_2}$ em sentido horário ou em sentido anti-horário em relação à sua extremidade comum p_0, simplesmente fazemos a conversão para usar p_0 como origem. Isto é, denotamos o vetor $p_1' = (x_1', y_1')$ por $p_1 - p_0$, onde $x_1' = x_1 - x_0$ e $y_1' = y_1 - y_0$, e definimos $p_2 - p_0$ de maneira semelhante. Então calculamos o produto cruzado

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0).$$

Se esse produto cruzado é positivo, então $\overrightarrow{p_0 p_1}$ está em sentido horário em relação a $\overrightarrow{p_0 p_2}$; se é negativo, ele está em sentido anti-horário.

Determinando se segmentos consecutivos viram para a esquerda ou para a direita

Nossa próxima pergunta é se dois segmentos de reta consecutivos $\overline{p_0 p_1}$ e $\overline{p_1 p_2}$ viram para a esquerda ou para a direita no ponto p_1. Equivalentemente, queremos um método para determinar para que lado se curva um ângulo $\angle p_0\, p_1\, p_2$ dado. Produtos cruzados nos permitem responder a essa pergunta sem calcular o ângulo. Como mostra a Figura 33.2, basta verificar se o segmento dirigido $\overrightarrow{p_0 p_2}$ está em sentido horário ou em sentido anti-horário em relação ao segmento dirigido $\overrightarrow{p_0 p_1}$. Para tal, calculamos o produto cruzado $(p_2 - p_0) \times (p_1 - p_0)$. Se o sinal desse produto cruzado é negativo, então $\overrightarrow{p_0 p_2}$ está em sentido anti-horário em relação a $\overrightarrow{p_0 p_1}$ e, portanto, viramos para a esquerda em p_1. Um produto cruzado positivo indica uma orientação horária e uma virada para a direita. Um produto cruzado igual a 0 significa que os pontos p_0, p_1 e p_2 são colineares.

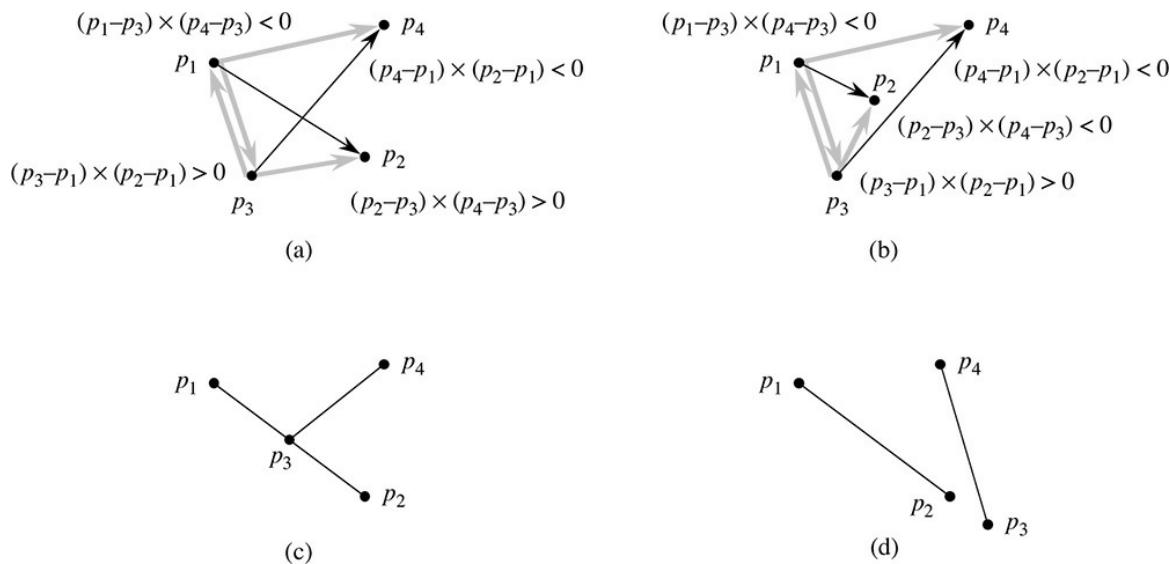
Determinando se dois segmentos de reta se interceptam

Para determinar se dois segmentos de reta se interceptam, verificamos se cada segmento ultrapassa a reta que contém o outro. Um segmento $\overline{p_0 p_2}$ ***ultrapassa*** uma reta se o ponto p_1 se encontra em um lado da reta e o ponto p_2 se encontra no outro lado. Um caso limite surge se p_1 ou p_2 se encontra sobre a reta. Dois segmentos de reta se interceptam se e somente se uma das condições a seguir é válida (ou ambas):

1. Cada segmento ultrapassa a reta que contém o outro.
2. Uma extremidade de um segmento encontra-se sobre o outro segmento. (Essa condição vem do caso limite.)

Figura 33.2 Usando o produto cruzado para determinar como segmentos de reta consecutivos p_0p_1 e p_1p_2 viram no ponto p_1. Verificamos se o segmento dirigido p_0p_2 está em sentido horário ou em sentido anti-horário em relação ao segmento dirigido p_0p_1. (**a**) Se em sentido anti-horário, os pontos viram para a esquerda. (**b**) Se em sentido horário, os pontos viram para a direita.

Os procedimentos a seguir implementam essa ideia. SEGMENTS-INTERSECT RETORNA TRUE se os segmentos p_1p_2 e p_3p_4 se interceptam e FALSE se eles não se interceptam. Eles chamam as sub-rotinas DIRECTION, que calcula orientações relativas usando o método do produto cruzado já descrito, e ON-SEGMENT, que determina se um ponto conhecido que sabemos ser colinear com um segmento encontra-se sobre esse segmento.

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
1  d1 = DIRECTION(p3, p4, p1)
2  d2 = DIRECTION(p3, p4, p2)
3  d3 = DIRECTION(p1, p2, p3)
4  d4 = DIRECTION(p1, p2, p4)
5  if ((d1 > 0 e d2 < 0) ou (d1 < 0 e d2 > 0)) e
        ((d3 > 0 e d4 < 0) ou (d3 < 0 e d4 > 0))
6    return TRUE
7  elseif d1 = 0 e ON-SEGMENT(p3, p4, p1)
8    return TRUE
9  elseif d2 == 0 e ON-SEGMENT(p3, p4, p2)
10   return TRUE
11 elseif d3 == 0 e ON-SEGMENT(p1, p2, p3)
12   return TRUE
13 elseif d4 == 0 e ON-SEGMENT(p1, p2, p4)
14   return TRUE
15 else return FALSE

DIRECTION(pi, pj, pk)
1 return (pk − pi) × (pj − pi)

ON-SEGMENT(pi, pj, pk)
1 if min(xi, xj) ≤ xk ≤ max(xi, xj) e min(yi, yj) ≤ yk ≤ max(yi, yj)
2   return TRUE
3 else return FALSE
```

SEGMENTS-INTERSECT funciona da maneira ilustrada a seguir. As linhas 1–4 calculam a direção relativa d_i de cada extremidade p_i em relação ao outro segmento. Se todas as direções relativas forem não nulas, então é fácil determinar se os segmentos p_1p_2 e p_3p_4 se interceptam, da maneira descrita a seguir. O segmento p_1p_2 ultrapassa a reta que

contém o segmento $p_3 p_4$ se os segmentos dirigidos $p_3 p_1$ e $p_3 p_2$ têm orientações opostas em relação a $p_3 p_4$. Nesse caso, os sinais de d_1 e d_2 são diferentes. De modo semelhante, o segmento $p_3 p_4$ ultrapassa a reta que contém $p_1 p_2$ se os sinais de d_3 e d_4 são diferentes. Se o teste da linha 5 é verdadeiro, então os segmentos ultrapassam um ao outro, e SEGMENTS-INTERSECT retorna TRUE. A Figura 33.3(a) mostra esse caso. Caso contrário, os segmentos não ultrapassam as retas de um e de outro, embora um caso limite possa ser aplicável. Se todas as orientações relativas forem não nulas, nenhum caso limite é aplicável. Então, todos os testes de comparação com 0 nas linhas 7–13 falham, e SEGMENTS-INTERSECT retorna FALSE na linha 15. A Figura 33.3(b) mostra esse caso.

Um caso limite ocorre se qualquer orientação relativa d_k é 0. Aqui, sabemos que p_k é colinear com o outro segmento. Ele está diretamente sobre o outro segmento se e somente se estiver entre as extremidades do outro segmento. O procedimento ON-SEGMENT retorna se p_k está entre as extremidades do segmento $p_1 p_j$, que será o outro segmento quando chamado nas linhas 7–13; o procedimento supõe que p_k é colinear com o segmento $p_1 p_j$. As Figuras 33.3(c) e (d) mostram casos com pontos colineares. Na Figura 33.3(c), p_3 está em $p_1 p_2$, e portanto SEGMENTS-INTERSECT retorna TRUE na linha 12. Nenhuma extremidade está sobre outros segmentos na Figura 33.3(d) e, portanto, SEGMENTS-INTERSECT retorna FALSE na linha 15.

Figura 33.3 Casos no procedimento SEGMENTS-INTERSECT. (**a**) Os segmentos $p_1 p_2$ e $p_3 p_4$ ultrapassam as retas um do outro. Como $p_3 p_4$ ultrapassa a reta que contém $p_1 p_2$, os sinais dos produtos cruzados $(p_3 - p_1) \times (p_2 - p_1)$ e $(p_4 - p_1) \times (p_2 - p_1)$ são diferentes. Como $p_1 p_2$ ultrapassa a reta que contém $p_3 p_4$, os sinais dos produtos cruzados $(p_1 - p_3) \times (p_4 - p_3)$ e $(p_2 - p_3) \times (p_4 - p_3)$ são diferentes. (**b**) O segmento $p_3 p_4$ ultrapassa a reta que contém $p_1 p_2$, mas $p_1 p_2$ não ultrapassa a reta que contém $p_3 p_4$. Os sinais dos produtos cruzados $(p_1 - p_3) \times (p_4 - p_3)$ e $(p_2 - p_3) \times (p_4 - p_3)$ são iguais. (**c**) O ponto p_3 é colinear com $p_1 p_2$ e está entre p_1 e p_2 . (**d**) O ponto p_3 é colinear com $p_1 p_2$, mas não está entre p_1 e p_2 . Os segmentos não se interceptam.

Outras aplicações de produtos cruzados

Seções posteriores deste capítulo apresentam utilizações adicionais para produtos cruzados. Na Seção 33.3, precisaremos ordenar um conjunto de pontos de acordo com seus ângulos polares em relação a uma origem dada. Como o Exercício 33.1-3 pede que você mostre, podemos usar produtos cruzados para executar as comparações no procedimento de ordenação. Na Seção 33.2, usaremos árvores vermelho-preto para manter a ordenação vertical de um conjunto de segmentos de reta. Em vez de manter valores de chaves explícitos que comparamos uns aos outros no código da árvore vermelho-preto, calcularemos um produto cruzado para determinar qual dos dois segmentos que interceptam uma dada reta vertical está acima do outro.

Exercícios

33.1-1 Prove que, se $p_1 \times p_2$ é positivo, então o vetor p_1 está em sentido horário em relação ao vetor p_2 em relação à origem (0, 0) e que, se seu produto cruzado é negativo, então p_1 está em sentido anti-horário em relação a p_2.

33.1-2 O professor van Pelt sugere que somente a dimensão x precisa ser testada na linha 1 de ON-SEGMENT. Mostre por que o professor está errado.

33.1-3 O ***ângulo polar*** de um ponto p_1 em relação a um ponto de origem p_0 é o ângulo do vetor $p_1 - p_0$ no sistema de coordenadas polares habitual. Por exemplo, o ângulo polar de (3, 5) em relação a (2, 4) é o ângulo do vetor (1, 1), que é 45 graus ou $p/4$ radianos. O ângulo polar de (3, 3) em relação a (2, 4) é o ângulo do vetor (1, –1), que é 315 graus ou $7p/4$ radianos. Escreva pseudocódigo para ordenar uma sequência $\langle p_1, p_2, ..., p_n \rangle$ de n pontos de acordo com seus ângulos polares em relação a um determinado ponto de origem p_0. Seu procedimento deve demorar o tempo $O(n \lg n)$ e usar produtos cruzados para comparar ângulos.

33.1-4 Mostre como determinar no tempo $O(n_2 \lg n)$ se três pontos quaisquer em um conjunto de n pontos são colineares.

33.1-5 Um ***polígono*** é uma curva fechada formada por segmentos lineares no plano. Ou seja, é uma curva que termina em si mesma e é formada por uma sequência de segmentos de reta denominados ***lados*** do polígono. Um ponto que une dois lados consecutivos é denominado ***vértice*** do polígono. Se o polígono é ***simples***, o que geralmente consideraremos aqui, seus lados não se cruzam. O conjunto de pontos no plano delimitados por um polígono simples forma o ***interior*** do polígono, o conjunto de pontos sobre o polígono propriamente dito forma seu ***contorno***, e o conjunto de pontos que cercam o polígono forma seu ***exterior***. Um polígono simples é ***convexo*** se, dados quaisquer dois pontos em seu contorno ou em seu interior, todos os pontos sobre o segmento de reta traçado entre eles estão contidos no contorno ou no interior do polígono. Um vértice de um polígono convexo não pode ser expresso como uma combinação convexa de quaisquer dois pontos sobre o contorno ou no interior do polígono.

O professor Amundsen propõe o método a seguir para determinar se uma sequência $\langle p_0, p_1, ..., p_{n-1} \rangle$ de n pontos forma os vértices consecutivos de um polígono convexo. O resultado é "sim" se o conjunto $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, ..., n - 1\}$, onde a adição de índices é executada módulo n, não contém ambas, curvas para a esquerda e curvas para a direita; caso contrário, o resultado é "não". Mostre que, embora esse método seja executado em tempo linear, nem sempre produz a resposta correta. Modifique o método do professor de modo que ele sempre produza a resposta correta em tempo linear.

33.1-6 Dado um ponto $p_0 = (x_0, y_0)$, o ***raio horizontal direito*** de p_0 é o conjunto de pontos $\{p_i = (x_i, y_i) : x_i \geq x_0$ e $y_i = y_0\}$, isto é, é o conjunto de pontos que devem estar à direita de p_0, juntamente com o próprio p_0. Mostre como determinar se um dado raio horizontal direito de p_0 intercepta um segmento $p_1 p_2$ de reta no tempo $O(1)$ reduzindo o problema ao determinar se dois segmentos de reta se interceptam.

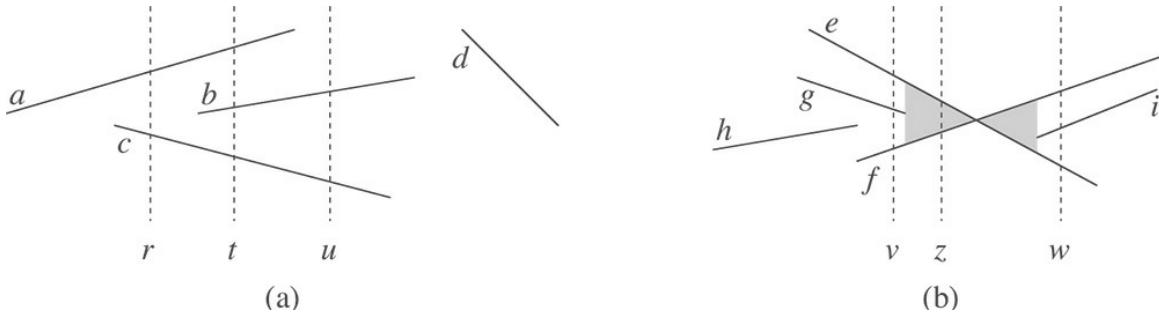
33.1-7 Um modo de determinar se um ponto p_0 está no interior de um polígono P simples, mas não necessariamente convexo, é observar qualquer raio de p_0 e verificar que o raio intercepta o contorno de P um número ímpar de vezes, mas que o próprio p_0 não está no contorno de P. Mostre como calcular no tempo $Q(n)$ se um ponto p_0 está no interior de um polígono P de n vértices. (*Sugestão*: Use o Exercício 33.1-6. Certifique-se de que seu algoritmo está correto quando o raio intercepta o contorno do polígono em um vértice e quando o raio se sobrepõe a um lado do polígono.)

33.1-8 Mostre como calcular a área de um polígono de n vértices simples, mas não necessariamente convexo, no tempo Q(n). (Veja no Exercício 33.1-5 definições pertinentes a polígonos.)

33.2 Determinando se dois segmentos quaisquer se interceptam

Esta seção apresenta um algoritmo para determinar se dois segmentos de reta quaisquer em um conjunto de segmentos se interceptam. O algoritmo utiliza uma técnica conhecida como "varredura", comum a muitos algoritmos de geometria computacional. Além disso, como mostram os exercícios no final desta seção, esse algoritmo, ou variações simples dele, pode ajudar a resolver outros problemas de geometria computacional.

O algoritmo é executado no tempo $O(n \lg n)$, onde n é o número de segmentos dados. Determina somente se existe ou não alguma interseção; não imprime todas as interseções. (Pelo Exercício 33.2-1, ele demora o tempo (n_2) no pior caso para encontrar *todas* as interseções em um conjunto de n segmentos de reta.)

Na ***varredura***, uma ***linha de varredura*** vertical imaginária passa pelo conjunto de objetos geométricos dado, em geral da esquerda para a direita. Tratamos a dimensão espacial pela qual a linha de varredura se move, nesse caso a dimensão x, como uma dimensão de tempo. A varredura nos dá um método para ordenar objetos geométricos, normalmente colocando-os em uma estrutura de dados dinâmica, e para aproveitar as relações entre eles. O algoritmo de interseção de segmentos de reta nesta seção considera todas as extremidades de segmentos de reta em ordem da esquerda para a direita e verifica se há uma interseção toda vez que encontra uma extremidade.

Para descrever e provar a correção de nosso algoritmo para determinar se dois segmentos de reta quaisquer entre n segmentos de reta se interceptam, adotaremos duas hipóteses simplificadoras. A primeira é que supomos que nenhum segmento de entrada é vertical. A segunda é que supomos que nenhuma tripla desses segmentos se intercepta em um único ponto. Os Exercícios 33.2-8 e 33.2-9 pedem para mostrar que o algoritmo é suficientemente robusto e por isso precisa somente de uma ligeira modificação para funcionar mesmo quando essas hipóteses não são válidas. Na realidade, eliminar tais hipóteses simplificadoras e lidar com condições de contorno, muitas vezes, configuram os desafios mais difíceis na programação de algoritmos de geometria computacional e na prova de sua correção.

Ordenar segmentos

Visto que supomos que não há nenhum segmento vertical, sabemos que qualquer segmento de entrada que intercepta uma linha de varredura vertical dada intercepta a linha de varredura em um único ponto. Assim, podemos ordenar os segmentos que interceptam uma linha de varredura vertical de acordo com as coordenadas y dos pontos de interseção.

Para sermos mais precisos, considere dois segmentos s_1 e s_2. Dizemos que esses segmentos são ***comparáveis*** em x se a linha de varredura vertical cuja coordenada x é x interceptar ambos os segmentos. Dizemos que s_1 está ***acima*** de s_2 em x, indicado por $s_1 \,^{x}\, s_2$, se s_1 e s_2 são comparáveis em x e a interseção de s_1 com a linha de varredura em x é mais alta que a interseção de s_2 com a mesma linha de varredura ou se s_1 e s_2 se interceptarem na linha de varredura. Na Figura 33.4(a), por exemplo, temos as relações $a \,^{r}\, c$, $a \,^{t}\, b$, $b^{t}\, c$, $a \,^{t}\, c$, e $b \,^{u}\, c$. O segmento d não é comparável com nenhum outro segmento.

Para qualquer x dado, a relação "^{x}" é uma pré-ordenação total (ver a Seção B.2) para todos os segmentos que interceptam a linha de varredura em x. Isto é, a relação é transitiva e, se cada um dos segmentos s_1 e s_2 interceptarem a linha de varredura em x, então $s_1 \,^{x}\, s_2$ ou $s_2 \,^{x}\, s_1$ ou ambos (se s_1 e s_2 se interceptam na linha de varredura). (A relação ^{x} é também reflexiva, mas não é simétrica nem assimétrica.) Todavia, a pré-ordenação total pode ser diferente para valores diferentes de x, à medida que segmentos entram e saem da ordenação. Um segmento entra na ordenação quando sua extremidade esquerda é encontrada pela varredura e sai da ordenação quando sua extremidade direita é encontrada.

Figura 33.4 Ordenação entre segmentos de reta em diversas linhas de varredura verticais. (**a**) Temos $a \,_r\, c$, $a \,_t\, b$, $b_t\, c$, $a \,_t\, c$, e $b \,_u\, c$. O segmento d não é comparável a nenhum outro segmento mostrado. (**b**) Quando os segmentos e e f se interceptam, suas ordens são invertidas: temos $e \,_v$, mas $f_w\, e$. Qualquer linha de varredura (tal como z) que passe pela região sombreada tem e e f consecutivos na ordenação dada pela relação z.

O que acontece quando a linha de varredura passa pela interseção de dois segmentos? Como mostra a Figura 33.4(b), as posições dos segmentos são invertidas na pré-ordenação total. As linhas de varredura v e w estão à esquerda e à direita, respectivamente, do ponto de interseção dos segmentos e e f, e temos $e \,^v f$ e $f \,^w e$. Observe que, como supomos que não há três segmentos que se interceptam no mesmo ponto, deve existir alguma linha de varredura vertical x para a qual os segmentos e e f que se interceptam são *consecutivos* na pré-ordenação total ^{x}. Qualquer linha de varredura que passe pela região sombreada da Figura 33.4(b), tal como z, tem e e f consecutivos em sua pré-ordenação total.

Movendo a linha de varredura

Os algoritmos de varredura normalmente administram dois conjuntos de dados:

1. O ***status da linha de varredura*** dá as relações entre os objetos que a linha de varredura intercepta.
2. O ***escalonamento de pontos eventuais*** é uma sequência de pontos denominados ***pontos eventuais*** ordenamos da esquerda para a direita, de acordo com suas coordenadas x. À medida que a varredura progride da esquerda para a direita, sempre que a linha de varredura alcança a coordenada x de um ponto eventual, a varredura para e depois recomeça. Mudanças no status da linha de varredura ocorrem somente em pontos eventuais.

Para alguns algoritmos (por exemplo, o algoritmo solicitado no Exercício 33.2-7), o escalonamento de pontos eventuais desenvolve-se dinamicamente à medida que o algoritmo progride. Entretanto, o algoritmo à mão determina todos os pontos eventuais antes da varredura, exclusivamente com base em propriedades simples dos dados de entrada. (Se duas ou mais extremidades são ***coverticais***, isto é, se tiverem a mesma coordenada x, desempatamos colocando todas as extremidades coverticais esquerdas antes das extremidades coverticais direitas. Dentro de um conjunto de extremidades esquerdas coverticais, colocamos em primeiro lugar as que têm coordenadas y mais baixas e fazemos o mesmo com um conjunto de extremidades coverticais direitas. Quando encontramos a extremidade esquerda de um segmento, inserimos o segmento no status da linha de varredura e eliminamos o segmento do status da linha de varredura quando encontramos sua extremidade direita. Sempre que dois segmentos se tornam consecutivos pela primeira vez na pré-ordenação total, verificamos se eles se interceptam.

O status da linha de varredura é uma pré-ordenação total T, para a qual exigimos as seguintes operações:

- INSERT(T, s): insere segmento s em T.
- DELETE(T, s): elimina o segmento s de T.
- ABOVE(T, s): retorna o segmento imediatamente acima do segmento s em T.
- BELOW(T, s): retorna o segmento imediatamente abaixo do segmento s em T.

É possível que os segmentos s_1 e s_2 estejam mutuamente um acima do outro na pré-ordenação total T; essa situação pode ocorrer se s_1 e s_2 se interceptam na linha de varredura cuja pré-ordenação total é dada por T. Nesse

caso, os dois segmentos podem aparecer em qualquer das ordens em T.

Se a entrada contiver n segmentos, podemos efetuar cada uma das operações Insert, Delete, Above e Below no tempo $O(\lg n)$ usando árvores vermelho-preto. Lembre-se de que as operações em árvores vermelho-preto vistas no Capítulo 13 envolvem a comparação de chaves. Podemos substituir as comparações de chaves por comparações que usam produtos cruzados para determinar a ordenação relativa de dois segmentos (veja o Exercício 33.2-2).

Pseudocódigo de interseção de segmentos

O algoritmo a seguir toma como entrada um conjunto S de n segmentos de reta e devolve o valor booleano True se qualquer par de segmentos em S se intercepta, caso contrário devolve False. A árvore vermelho-preto mantém a pré-ordenação total T.

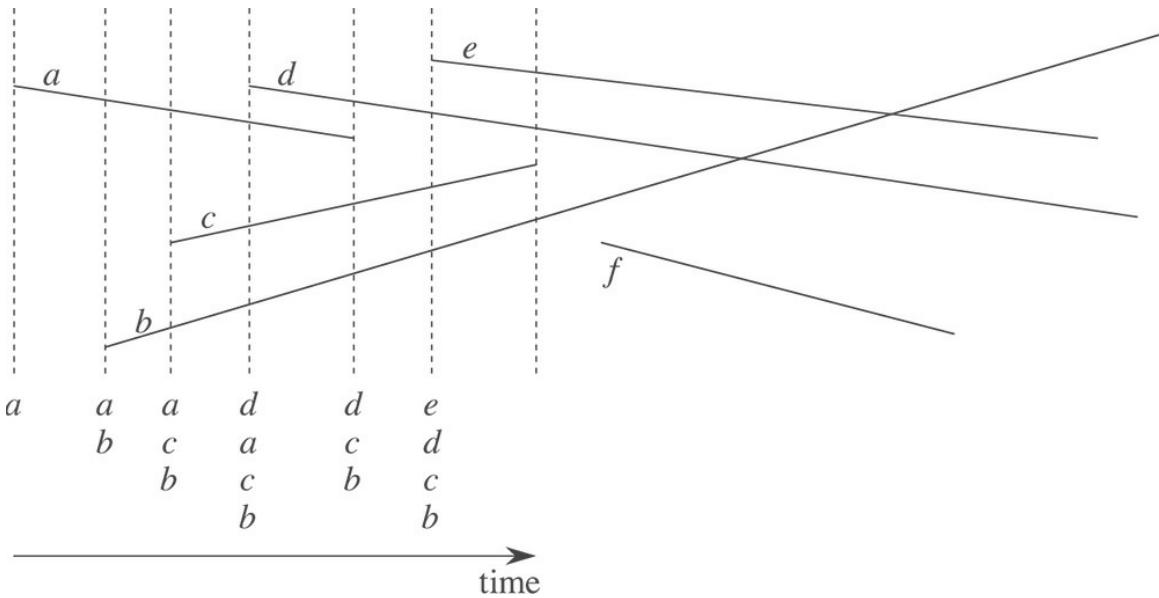
```
ANY-SEGMENTS-INTERSECT(S)
1  T = ∅
2  ordenar as extremidades dos segmentos em S da esquerda para a direita,
       resolvendo empates colocando extremidades esquerdas
       antes de extremidades direitas e resolvendo empates adicionais colocando em primei-
       ro lugar pontos com coordenadas y mais baixas
3  for cada ponto p na lista ordenada de extremidades
4      if p é a extremidade esquerda de um segmento s
5          INSERT(T, s)
6              if (ABOVE(T, s) existe e intercepta s)
                       ou (BELOW(T, s) existe e intercepta s)
7                  return TRUE
8        if p é a extremidade direita de um segmento s
9            if ABOVE(T, s) e BELOW(T, s) existem
                           e ABOVE(T, s) intercepta BELOW(T, s)
10                   return TRUE
11               DELETE(T, s)
12  return FALSE
```

A Figura 33.5 ilustra como o algoritmo funciona. A linha 1 inicializa a pré-ordenação total como vazia. A linha 2 determina o escalonamento de pontos eventuais ordenando as $2n$ extremidades de segmentos da esquerda para a direita, resolvendo empates da maneira já descrita. Um modo de executar a linha 2 é ordenar lexicograficamente as extremidades em (x, e, y), onde x e y são as coordenadas habituais, $e = 0$ para uma extremidade esquerda e $e = 1$ para uma extremidade direita.

Figura 33.5 Execução de Any-segments-intersect. Cada linha tracejada é a linha de varredura em um ponto eventual. Exceto na linha de varredura da extrema direita, a ordenação de nomes de segmentos abaixo de cada linha de varredura corresponde à pré-ordenação total T no final do processamento do laço **for** para o ponto eventual correspondente. A linha de varredura da extrema direita ocorre durante o processamento da extremidade direita do segmento c; como os segmentos d e b circundam c e se interceptam, o procedimento devolve TRUE.

Cada iteração do laço **for** das linhas 3–11 processa um ponto eventual p. Se p é a extremidade esquerda de um segmento s, a linha 5 acrescenta s à pré-ordenação total, e as linhas 6–7 devolvem TRUE se s intercepta qualquer dos segmentos aos quais é consecutivo na pré-ordenação total definida pela linha de varredura que passa por p. (Uma condição de contorno ocorre se p encontra-se sobre um outro segmento s'. Nesse caso, basta que s e s' sejam inseridos consecutivamente em T.) Se p é a extremidade direita de um segmento s, então precisamos eliminar s da pré-ordenação total. Mas, primeiro, as linhas 9–10 devolvem TRUE se existe uma interseção entre os segmentos que circundam s na pré-ordenação total definida pela linha de varredura que passa por p. Se esses segmentos não se interceptam, a linha 11 elimina o segmento s da pré-ordenação total. Se os segmentos que circundam o segmento s se interceptam, eles teriam se tornado consecutivos após a eliminação de s, caso a declaração **return na linha** 10 não tivesse impedido a execução da linha 11. O argumento de correção, que apresentamos em seguida, esclarecerá por que é suficiente verificar os segmentos que circundam s. Finalmente, se nunca encontrarmos, uma interseção após o processamento de todos os $2n$ pontos eventuais, a linha 12 devolverá False.

Correção

Para mostrar que Any-Segments-Intersect é correto, provaremos que a chamada Any-Segments-Intersect(S) devolve True se e somente se existe uma interseção entre os segmentos em S.

É fácil ver que Any-Segments-Intersect devolve True (nas linhas 7 e 10) somente se encontra uma interseção entre dois dos segmentos de entrada. Consequentemente, se ele retorna True, existe uma interseção.

Também precisamos mostrar o inverso: se existe uma interseção, então Any-Segments-Intersect devolve True. Vamos supor que haja no mínimo uma interseção. Seja p o ponto de interseção na mais à esquerda, decidindo empates escolhendo o ponto que tem a menor coordenada y, e sejam a e b os segmentos que se interceptam em p. Visto que não ocorre nenhuma interseção à esquerda de p, a ordem dada por T é correta em todos os pontos à esquerda de p. Como não há três segmentos que se interceptam no mesmo ponto, a e b se tornam consecutivos na pré-ordenação total em alguma linha de varredura z.[2] Além disso, z está à esquerda de p ou passa por p. Alguma extremidade de segmento q na linha de varredura z é o ponto eventual no qual a e b se tornam consecutivos na pré-ordenação total. Se p está na

linha de varredura z, então $q = p$. Se p não está na linha de varredura z, então q está à esquerda de p. Em qualquer caso, a ordem dada por T é correta imediatamente antes de encontrar q. (É aqui que usamos a ordem lexicográfica na qual o algoritmo processa pontos eventuais. Como p é o mais baixo dos pontos de interseção da extrema esquerda, mesmo que p esteja sobre a linha de varredura z e algum outro ponto de interseção p' esteja sobre z, o ponto eventual $q = p$ é processado antes de a outra interseção p' poder interferir na pré-ordenação total T. Além disso, ainda que p seja a extremidade esquerda de um segmento, digamos a, e a extremidade direita do outro segmento seja, digamos b, como os eventos da extremidade esquerda ocorrem antes dos eventos da extremidade direita, o segmento b está em T quando encontra pela primeira vez o segmento a.) O ponto eventual q é processado por Any-Segments-Intersect ou não é processado.

Se q é processado por Any-Segments-Intersect, só há duas ações possíveis que podem ocorrer:

1. Ou a ou b é inserido em T, e o outro segmento está acima ou abaixo dele na pré-ordenação total. As linhas 4–7 detectam esse caso.
2. Os segmentos a e b já estão em T, e um segmento entre eles na pré-ordenação total é eliminado, o que transforma a e b em consecutivos. As linhas 8–11 detectam esse caso.

Em qualquer dos casos, encontramos a interseção p, e Any-Segments-Intersect devolve True.

Se o ponto eventual q não é processado por Any-Segments-Intersect, o procedimento deve ter retornado antes de processar todos os pontos eventuais. Essa situação só poderia ter ocorrido se Any-Segments-Intersect já tivesse encontrado uma interseção e devolvido True.

Assim, se existe uma interseção, Any-Segments-Intersect retorna True. Como já vimos, se Any-Segments-Intersect devolve True, existe uma interseção. Portanto, Any-Segments-Intersect sempre devolve uma resposta correta.

Tempo de execução

Se o conjunto S contém n segmentos, então Any-Segments-Intersect é executado no tempo $O(n \lg n)$. A linha 1 demora o tempo $O(1)$. A linha 2 demora o tempo $O(n \lg n)$, usando a ordenação por intercalação ou heapsort. O laço **for** das linhas 3–11 itera no máximo uma vez por ponto eventual e, portanto, com $2n$ pontos eventuais, o laço itera no máximo $2n$ vezes. Cada iteração demora o tempo $O(\lg n)$, já que cada operação da árvore vermelho-preto demora o tempo $O(\lg n)$ e, usando o método da Seção 33.1, cada teste de interseção demora o tempo $O(1)$. Assim, o tempo total é $O(n \lg n)$.

Exercícios

33.2-1 Mostre que um conjunto de n segmentos de reta pode conter Q(n_2) interseções.

33.2-2 Dados dois segmentos a e b que são comparáveis em x, mostre como determinar no tempo $O(1)$ qual dentre a ^{x} b ou b ^{x} a é válida. Suponha que nenhum dos segmentos é vertical. (*Sugestão:* Se a e b não se interceptam, você pode simplesmente usar produtos cruzados. Se a e b se interceptam — o que, é claro, se pode determinar usando apenas produtos cruzados — você ainda pode usar somente adição, subtração e multiplicação, evitando a divisão. É claro que, na aplicação da relação ^{x} utilizada aqui, se a e b se interceptam, podemos apenas parar e declarar que encontramos uma interseção.)

33.2-3 O professor Mason sugere que modifiquemos Any-Segments-Intersect de modo que, em vez de retornar ao encontrar uma interseção, imprima os segmentos que se interceptam e passe para a próxima iteração do laço **for**. O professor denomina o procedimento resultante Print-Intersecting-Segments e afirma que ele imprime todas as interseções, da esquerda para a direita, à medida que elas ocorrem no conjunto de segmentos de reta. O professor Dixon discorda e afirma que a ideia do professor Mason é incorreta. Qual dos professores está certo? O procedimento Print-Intersecting-Segments sempre encontra primeiro a interseção da extrema esquerda? Sempre encontrará todas as interseções?

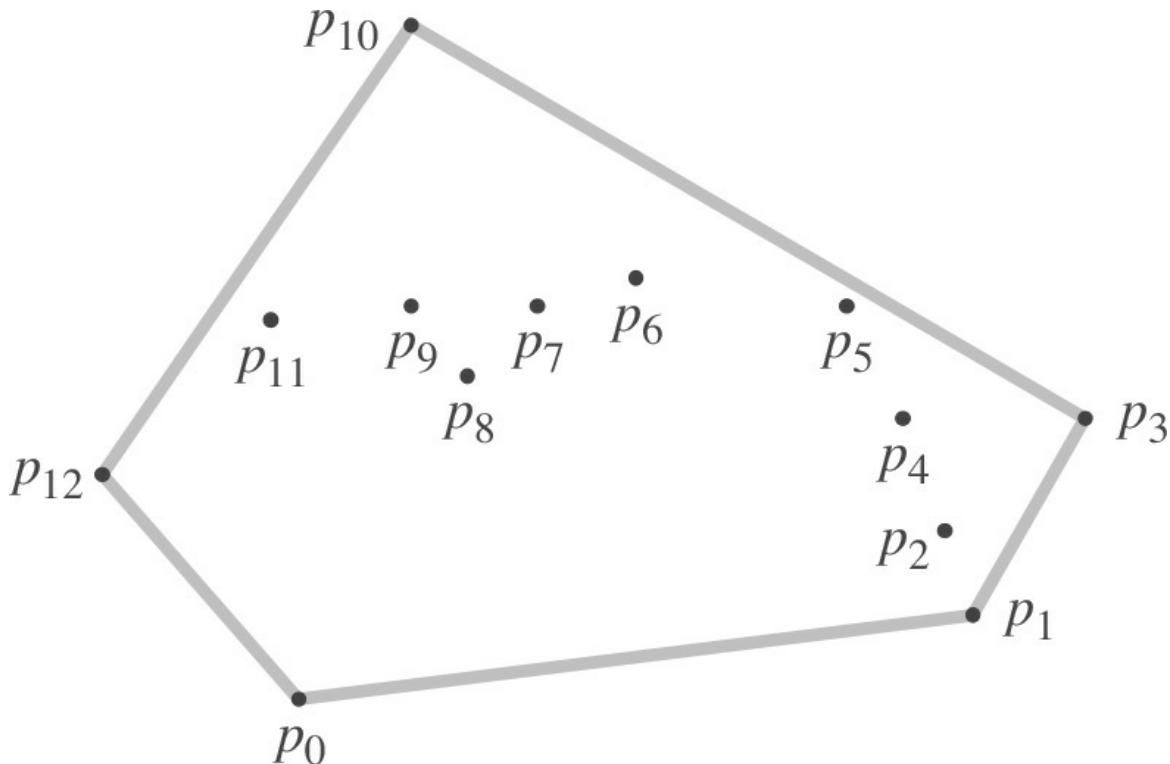
33.2-4 Dê um algoritmo de tempo $O(n \lg n)$ para determinar se um polígono de n vértices é simples.

33.2-5 Dê um algoritmo de tempo $O(n \lg n)$ para determinar se dois polígonos simples com um total de n vértices se interceptam.

33.2-6 Um ***disco*** consiste em uma circunferência mais seu interior e é representado por seu ponto central e raio. Dois discos se interceptam se têm algum ponto em comum. Dê um algoritmo de tempo $O(n \lg n)$ para determinar se dois discos quaisquer em um conjunto de n discos se interceptam.

33.2-7 Dado um conjunto de n segmentos de reta contendo um total de k interseções, mostre como apresentar todas as k interseções no tempo $O((n + k)\lg n)$.

33.2-8 Mostre que ANY-SEGMENTS-INTERSECT funciona corretamente mesmo que três ou mais segmentos se interceptem no mesmo ponto.

33.2-9 Mostre que ANY-SEGMENTS-INTERSECT funciona corretamente na presença de segmentos verticais se tratarmos a extremidade inferior de um segmento vertical como se fosse uma extremidade esquerda e a extremidade superior como se fosse uma extremidade direita. Como sua resposta ao Exercício 33.2-2 muda se permitirmos segmentos verticais?

33.3 DETERMINANDO A ENVOLTÓRIA CONVEXA

A ***envoltória convexa*** de um conjunto Q de pontos, denotada por CH(Q), é o menor polígono convexo P para o qual cada ponto em Q está no contorno de P ou em seu interior (veja no Exercício 33.1-5 uma definição precisa de polígono convexo). Supomos implicitamente que todos os pontos no conjunto Q são únicos e que Q contém no mínimo três pontos que não são colineares. Intuitivamente, podemos imaginar cada ponto em Q como um prego cuja cabeça fica um pouco acima de uma tábua. Então, a envoltória convexa é a forma produzida por uma tira elástica apertada que passa por todos os pregos. A Figura 33.6 mostra um conjunto de pontos e sua envoltória convexa.

Nesta seção, apresentaremos dois algoritmos que calculam a envoltória convexa de um conjunto de n pontos. Ambos produzem os vértices da envoltória em ordem anti-horária. O primeiro, conhecido como varredura de Graham, é executado no tempo $O(n \lg n)$. O segundo, denominado marcha de Jarvis, é executado no tempo $O(nh)$, onde h é o número de vértices da envoltória convexa. Como ilustra a Figura 33.6, todo vértice de CH(Q) é um ponto em Q. Ambos os algoritmos exploram essa propriedade, decidindo quais vértices em Q são mantidos como vértices da envoltória convexa e quais vértices em Q são descartados.

Podemos calcular envoltórias convexas no tempo $O(n \lg n)$ por qualquer um de vários métodos. A varredura de Graham e a marcha de Jarvis usam uma técnica denominada "varredura rotacional" e processam vértices na ordem dos ângulos polares que eles formam com um vértice de referência. Entre outros métodos, citamos os seguintes:

- No ***método incremental***, em primeiro lugar ordenamos os pontos da esquerda para a direita, produzindo uma sequência $\langle p_1, p_2, ..., p_n \rangle$. Na i-ésima etapa, atualizamos a envoltória convexa dos $i - 1$ pontos mais à esquerda, CH($\{p_1, p_2, ..., p_{i-1}\}$) de acordo com o i-ésimo ponto contando da esquerda, formando assim CH($\{p_1, p_2, ..., p_i\}$). O Exercício 33.3-6 pede que você mostre como implementar esse método de modo que seja executado no tempo total de $O(n \lg n)$.

Figura 33.6 Um conjunto de pontos $Q = \{p_0, p_1, ..., p_{12}\}$ com sua envoltória convexa CH(Q) traçada em cinzento.

- No método de ***divisão e conquista*** dividimos, no tempo Q(n), o conjunto de n pontos em dois subconjuntos, um contendo os n/2 pontos mais à esquerda e um contendo os n/2 pontos mais à direita, calculamos recursivamente as envoltórias convexas dos subconjuntos e depois, por meio de um método inteligente, combinamos as envoltórias no tempo Q(n). O tempo de execução é descrito pela conhecida recorrência $T(n) = 2T(n/2) + O(n)$, e, portanto, o método de divisão e conquista é executado no tempo $O(n \lg n)$.
- O ***método de poda e busca*** é semelhante ao algoritmo da mediana de tempo linear do pior caso da Seção 9.3. Com esse método, encontramos a porção superior (ou "cadeia superior") da envoltória convexa descartando repetidamente uma fração constante dos pontos remanescentes até restar somente a cadeia superior da envoltória convexa. Então, fazemos o mesmo para a cadeia inferior. Esse método é assintoticamente o mais rápido: se a envoltória convexa contém h vértices, ele é executado no tempo de somente $O(n \lg h)$.

Calcular a envoltória convexa de um conjunto de pontos é um problema interessante por si só. Além disso, algoritmos para alguns outros problemas de geometria computacional começam pelo cálculo de uma envoltória convexa. Considere, por exemplo,o ***problema do par mais afastado*** bidimensional: temos um conjunto de n pontos no plano e desejamos encontrar os dois pontos cuja distância entre eles seja máxima. Como o Exercício 33.3-3 pede para provar, esses dois pontos devem ser vértices da envoltória convexa. Se bem que não provaremos isso aqui, podemos encontrar o par de vértices mais afastado de um polígono convexo de n vértices no tempo $O(n)$. Assim, calculando a envoltória convexa dos n pontos de entrada no tempo $O(n \lg n)$ e depois encontrando o par mais afastado dos vértices do polígono convexo resultante, podemos localizar o par de pontos mais afastados em qualquer conjunto de n pontos no tempo $O(n \lg n)$.

Varredura de Graham

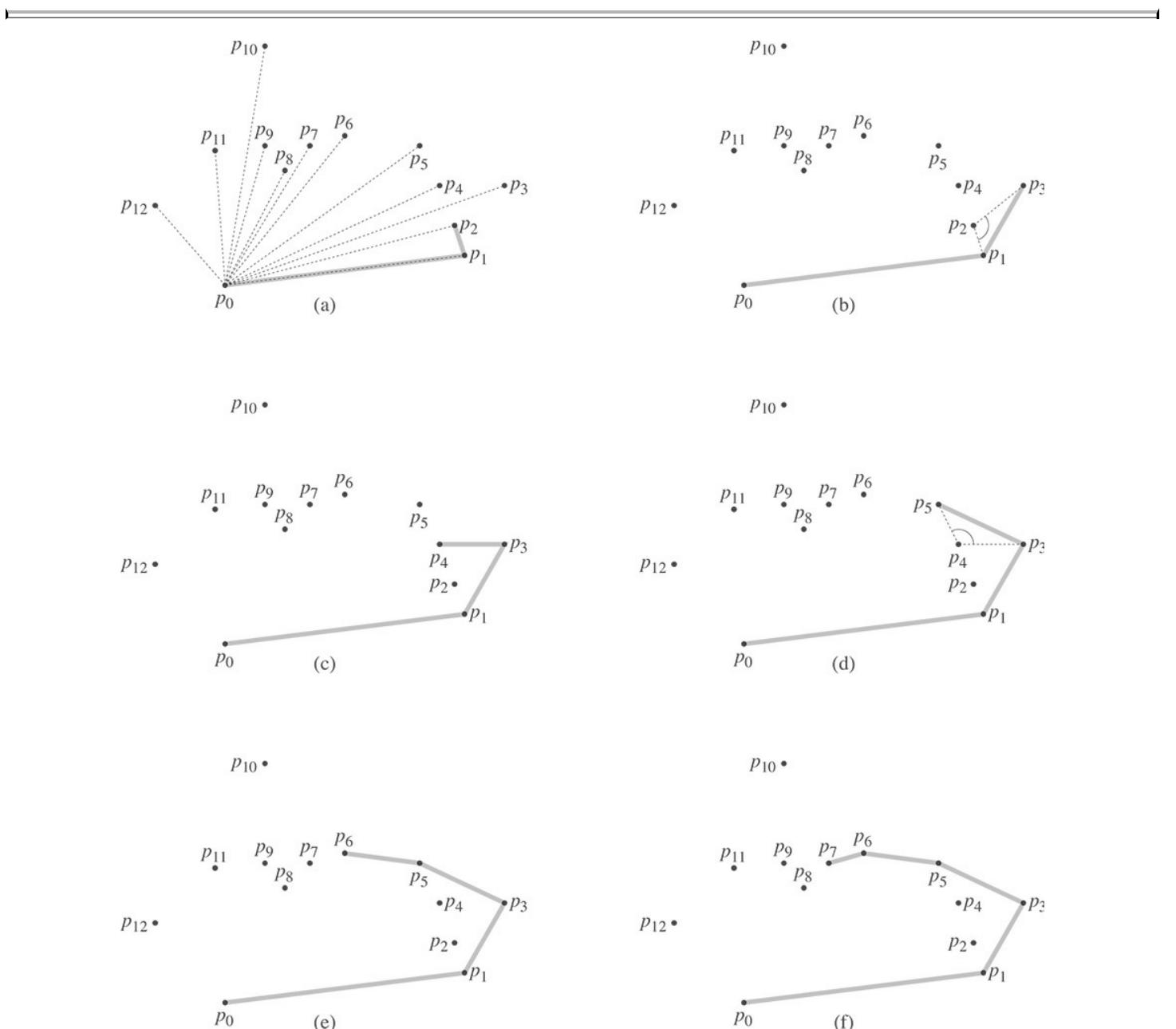
A ***varredura de Graham*** resolve o problema da envoltória convexa mantendo uma pilha S de pontos candidatos. A varredura insere cada ponto do conjunto de entrada Q na pilha uma vez e, a certa altura, retira da pilha cada ponto

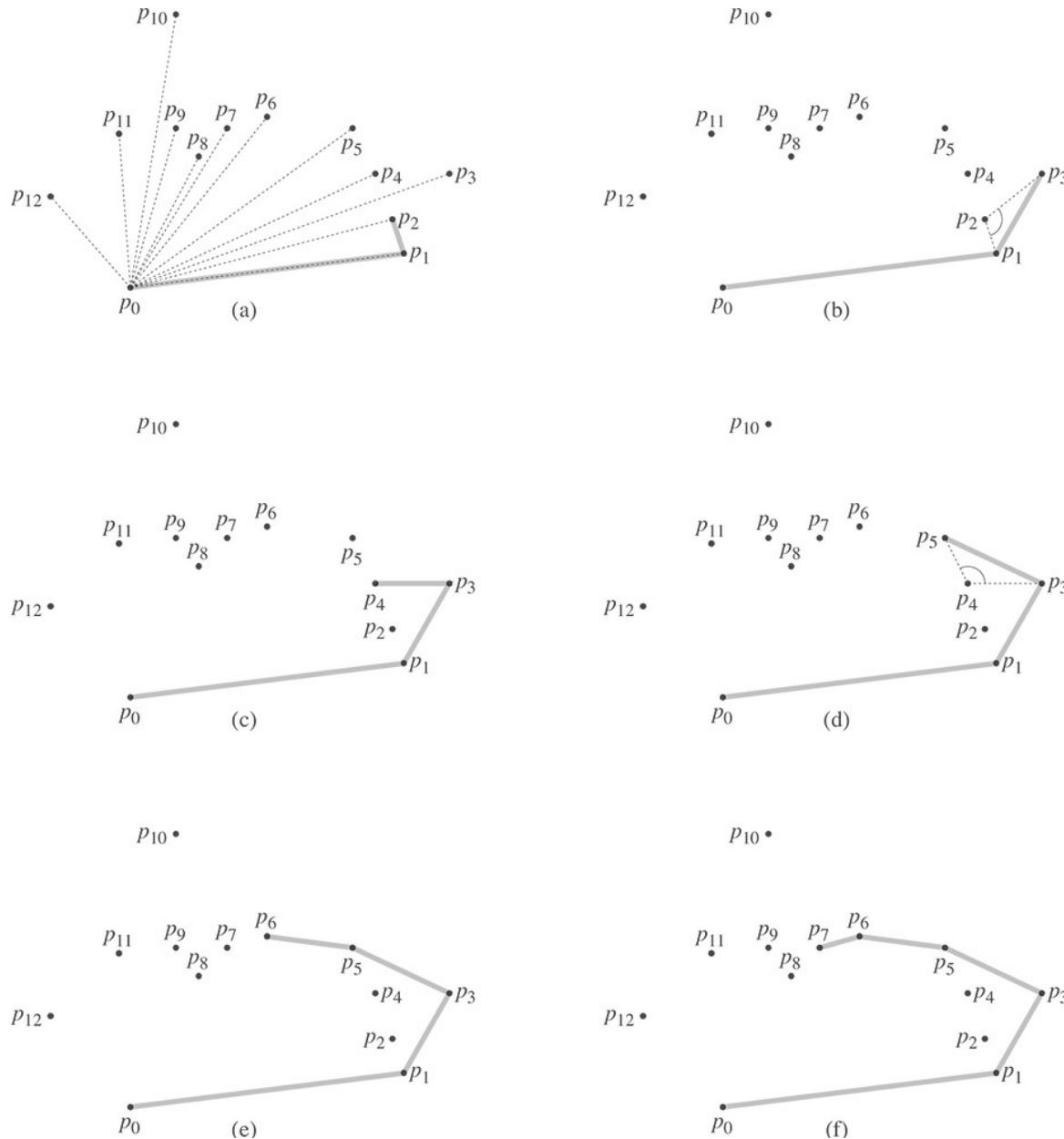
que não é um vértice de CH(Q). Quando o algoritmo termina, a pilha S contém exatamente os vértices de CH(Q), em ordem anti-horária das posições que ocupam na envoltória.

O procedimento GRAHAM-SCAN toma como entrada um conjunto Q de pontos, onde $|Q| \geq 3$. Chama a função TOP(S), que retorna o ponto que está no topo da pilha S sem mudar S, e a função NEXT-TO-TOP(S), que retorna o ponto que está uma entrada abaixo do topo da pilha S, sem mudar S. Como demonstraremos em breve, a pilha S retornada por GRAHAM-SCAN contém, de baixo para cima, exatamente os vértices de CH(Q) em ordem anti-horária.

```
GRAHAM-SCAN(Q)
 1  seja p0 o ponto em Q com a coordenada y mínima
        ou tal ponto que esteja mais à esquerda no caso de empate
 2  sejam ⟨p1, p2, ..., pm⟩ os pontos restantes em Q,
          ordenados por ângulo polar em ordem anti-horária em torno de p0
          (se mais de um ponto tiver o mesmo ângulo, remover todos eles, exceto o mais
          afastado de p0)
 3  seja S uma pilha vazia
 4  PUSH(p0, S)
 5  PUSH(p1, S)
 6  if m ≥ 2 PUSH(p2, S)
 7  for i = 3 to m
 8     while o ângulo formado pelos pontos NEXT-TO-TOP(S),
                 TOP(S) e pi curva não vira para a esquerda
 9           POP(S)
10        PUSH(pi, S)
11  return S
```

A Figura 33.7 ilustra o progresso de GRAHAM-SCAN. A linha 1 escolhe o ponto p_0 como o ponto que tem a coordenada y mais baixa e escolhe o ponto da extrema esquerda no caso de um empate. Visto que não existe nenhum ponto em Q que esteja abaixo de p_0 e todos os outros pontos com a mesma coordenada y estão à sua direita, p_0 deve ser um vértice de CH(Q). A linha 2 ordena os pontos restantes de Q por ângulo polar em relação a p_0 usando o mesmo método — comparação de produtos cruzados —, como no Exercício 33.1-3. Se dois ou mais pontos têm o mesmo ângulo polar em relação a p_0, todos exceto o mais afastado desses pontos são combinações convexas de p_0 e do ponto mais afastado e, portanto, não nos preocupamos mais com eles. Representamos por m o número de pontos que restam, exceto p_0. O ângulo polar, medido em radianos, de cada ponto de Q em relação a p_0 está no intervalo meio aberto $[0, p)$. Visto que os pontos são ordenados de acordo com ângulos polares, são ordenados em ordem anti-horária em relação a p_0. Designamos essa sequência ordenada de pontos por $\langle p_1, p_2, ..., p_m \rangle$. Observe que os pontos p_1 e p_m são vértices de CH(Q) (veja o Exercício 33.3-1). A Figura 33.7(a) mostra os pontos da Figura 33.6 numerados sequencialmente em ordem crescente de ângulo polar em relação a p_0.

O restante do procedimento utiliza a pilha S. As linhas 3–5 inicializam a pilha para conter, de baixo para cima, os três primeiros pontos p_0, p_1 e p_2. A Figura 33.7(a) mostra a pilha inicial S. O laço **for** das linhas 7–10 itera uma vez para cada ponto na subsequência $\langle p_3, p_4, ..., p_m \rangle$. Veremos que, após o processamento do ponto p_i, a pilha S contém, de baixo para cima, os vértices de CH$\langle p_0, p_1, ..., p_i \rangle$ em ordem anti-horária. O laço **while** das linhas 8–9 remove pontos da pilha se constatarmos que eles não são vértices da envoltória convexa. Quando percorremos a envoltória convexa em ordem anti-horária, devemos virar para a esquerda em cada vértice. Assim, toda vez que o laço **while** encontra um vértice no qual fazemos uma curva que não é para a esquerda, extraímos o vértice da pilha. (Como verifica virada que não é para a esquerda, em vez de verificar apenas uma virada para a direita, esse teste exclui a possibilidade de um ângulo raso (180º) em um vértice da envoltória convexa resultante. Não queremos nenhum ângulo raso, já que nenhum vértice de um polígono convexo pode ser uma combinação convexa de outros vértices do polígono.) Depois de extrair todos os vértices que têm viradas que não são para a esquerda quando seguimos na direção do ponto p_i, inserimos p_i

na pilha. As Figuras 33.7 (b)–(k) mostram o estado da pilha S após cada iteração do laço **for**. Finalmente, GRAHAM-SCAN devolve a pilha S na linha 11. A Figura 33.7(l) mostra a envoltória convexa correspondente.

O teorema a seguir prova formalmente a correção de GRAHAM-SCAN.

Teorema 33.1 (Correção da varredura de Graham)

Se GRAHAM-SCAN é executado em um conjunto Q de pontos, onde $|Q| \geq 3$, então, no término, a pilha S consiste, de baixo para cima, exatamente nos vértices de CH(Q) em ordem anti-horária.

Prova Depois da linha 2, temos a sequência de pontos $\langle p_1, p_2, ..., p_m \rangle$. Vamos definir, para $i = 2, 3, ..., m$, o subconjunto de pontos $Q_i = \{p_0, p_1, ..., p_i\}$. Os pontos em $Q - Q_m$ são aqueles que foram removidos porque tinham o mesmo ângulo polar em relação a p_0 que algum ponto em Q_m; esses pontos não estão em CH(Q) e, portanto, CH(Q_m) = CH(Q). Assim, basta mostrar que, quando GRAHAM-SCAN termina, a pilha S consiste nos vértices de CH(Q_m) em ordem anti-horária, quando apresentados de baixo para cima. Observe que, exatamente como p_0, p_1 e pm são vértices de CH(Q), os pontos p_0, p_1 e p_i são vértices de CH(Q_i).

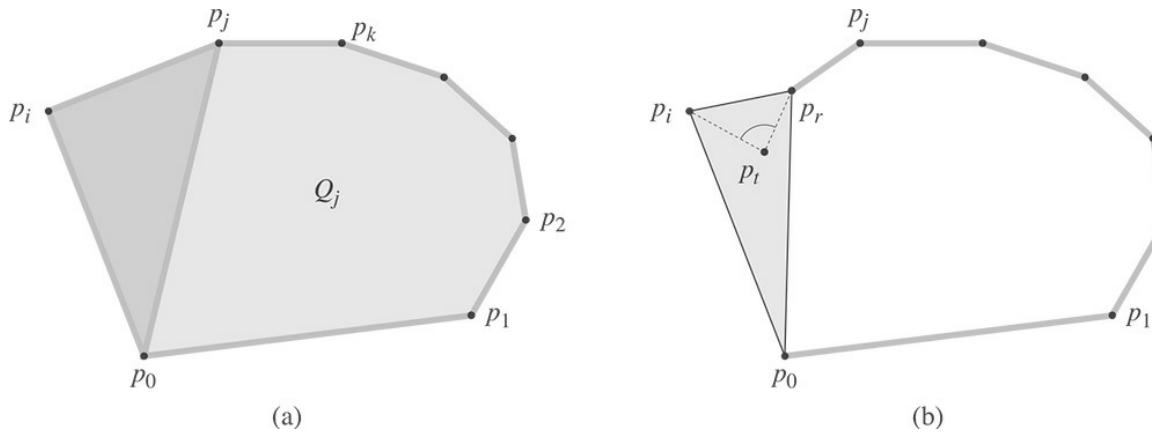
Figura 33.7 Execução de GRAHAM-SCAN no conjunto Q da Figura 33.6. A envoltória convexa atual contida na pilha S é mostrada em cinzento a cada etapa. (**a**) A sequência $\langle p_1, p_2, ..., p_{12} \rangle$ de pontos numerados em ordem crescente de ângulo polar em relação a p_0 e a pilha inicial S contendo p_0, p_1 e p_2. (**b**)–(**k**) A pilha S depois de cada iteração do laço **for** das linhas 7–10. Linhas tracejadas mostram viradas que não são para a esquerda, o que provoca a extração de pontos da pilha. Por exemplo, na parte (h) a virada para a direita no ângulo $\angle p_7 p_8 p_9$ resulta na extração de p_8 e, então, a virada para a direita no ângulo $p_6 p_7 p_9$ resulta na extração de p_7.

Figura 33.7 (continuação). (**l**) Envoltória convexa devolvida pelo procedimento, que corresponde à da Figura 33.6.

A prova utiliza o seguinte invariante de laço:

No início de cada iteração do laço **for** das linhas 7–10, a pilha S consiste, de baixo para cima, exatamente nos vértices de $CH(Q_{i-1})$ em ordem anti-horária.

Inicialização: O invariante é válido na primeira vez que executamos a linha 7, já que, nesse momento, a pilha S consiste exatamente nos vértices de $Q_2 = Q_{i-1}$, e esse conjunto de três vértices forma sua própria envoltória convexa. Além disso, eles aparecem em ordem anti-horária de baixo para cima.

Manutenção: Ao entrar em uma iteração do laço **for**, o ponto que está no alto da pilha S é p_{i-1}, que foi inserido no final da iteração anterior (ou antes da primeira iteração, quando $i = 3$). Seja p_j o ponto que está no alto da pilha S após a execução do laço **while** nas linhas 8–9, mas antes de a linha 10 inserir p_i, e seja p_k o ponto imediatamente abaixo de p_j em S. No momento em que p_j é o ponto que está no alto da pilha S e ainda não empurramos p_i, a pilha S contém exatamente os mesmos pontos que continha depois da iteração j do laço **for**. Então, pelo invariante de laço, S contém exatamente os vértices de $CH(Q_j)$ naquele momento e eles aparecem em ordem anti-horária de baixo para cima.

Vamos continuar a focalizar esse momento imediatamente antes de p_i ser empurrado. Sabemos que o ângulo polar de p_i em relação a p_0 é maior que o ângulo polar de p_j e que o ângulo $\angle p_k p_j p_i$ vira para a esquerda (caso contrário teríamos extraído p_j). Portanto, como S contém exatamente os vértices de $CH(Q_j)$, vemos, pela Figura 33.8(a), que, uma vez empurrado p_i, a pilha S conterá exatamente os vértices de $CH(Q_j \cup \{p_i\})$, ainda em ordem anti-horária de baixo para cima.

Agora, mostramos que $CH(Q_j \cup \{p_i\})$ é o mesmo conjunto de pontos que $CH(Q_i)$. Considere qualquer ponto p_t que tenha sido extraído durante a iteração i do laço **for** e seja p_r o ponto imediatamente abaixo de p_t na pilha S no momento em que p_t foi extraído (p_r poderia ser p_j). O ângulo $p_r p_t p_i$ faz uma curva não para a esquerda e o ângulo polar de p_t em relação a p_0 é maior que o ângulo polar de p_r. Como mostra a Figura 33.8(b), p_t deve estar no interior do triângulo formado por $\angle p_0$, p_r e p_i ou sobre um lado desse triângulo (mas ele não é um vértice do triângulo). É claro que, como p_t está dentro de um triângulo formado por três outros pontos de Q_i, ele não pode ser um vértice de $CH(Q_i)$. Visto que p_t não é um vértice de $CH(Q_i)$, temos que

$$CH(Q_i - \{p_t\}) = CH(Q_i) . \tag{33.1}$$

Seja P_i o conjunto de pontos que foram extraídos durante a iteração i do laço **for**. Visto que a igualdade (33.1) se aplica a todos os pontos em P_i, podemos aplicá-la repetidamente para mostrar que $CH(Q_i - P_i) = CH(Q_i)$. Porém, $Q_i - P_i = Q_j \cup \{p_i\}$ e, assim, concluímos que $CH(Q_j \{p_i\}) = CH(Q_i - P_i) = CH(Q_i)$.

Mostramos que, uma vez empurrado p_i, a pilha S contém exatamente os vértices de $CH(Q_i)$ em ordem anti-horária de baixo para cima. Então, incrementar i tornará o invariante de laço válido para a próxima iteração.

Figura 33.8 Prova da correção de GRAHAM-SCAN. (**a**) Como o ângulo polar de p_i em relação a p_0 é maior que o ângulo polar de p_j, e como o ângulo $p_k p_j p_i$ vira à esquerda, somar p_i a $CH(Q_j)$ resulta exatamente nos vértices de $CH(Q_j \cup \{pi\})$. (**b**) Se o ângulo $\angle p_r p_t p_i$ não vira para a esquerda, então p_t está no interior do triângulo formado por p_0, p_r e p_i ou sobre um lado do triângulo, o que significa que não pode ser um vértice de $CH(Q_i)$.

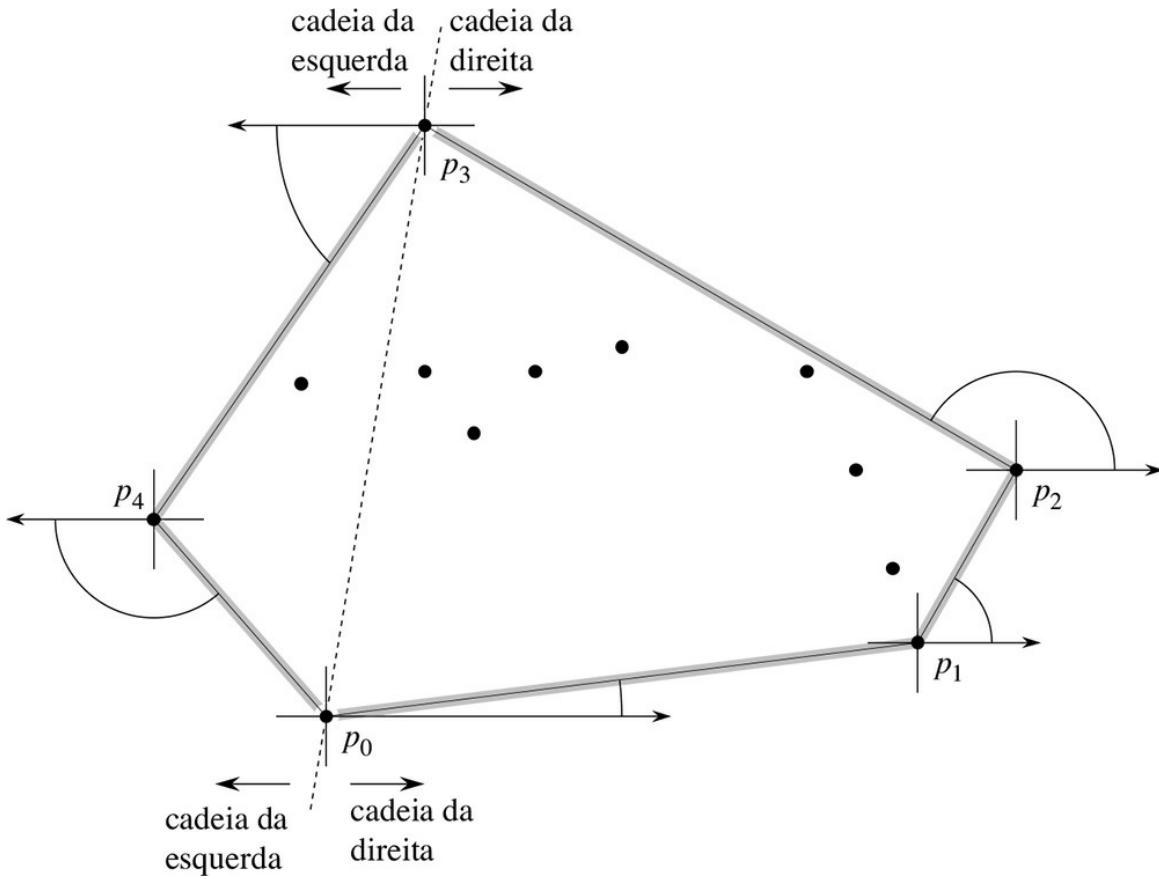
Término: Quando o laço termina, temos $i = m + 1$ e, portanto, o invariante de laço implica que a pilha S consiste exatamente nos vértices de CH(Q_m), que é CH(Q), em ordem anti-horária de baixo para cima. Isso conclui a prova.

Agora, mostraremos que o tempo de execução de GRAHAM-SCAN é $O(n \lg n)$, onde $n = |Q|$. A linha 1 demora o tempo Q(n). A linha 2 demora o tempo $O(n \lg n)$, usando ordenação por intercalação ou heapsort para ordenar os ângulos polares e o método de produtos cruzados da Seção 33.1 para comparar ângulos. (Podemos eliminar todos os pontos, exceto o mais afastado, que tenham o mesmo ângulo polar no tempo total $O(n)$ para todos os n pontos.) As linhas 3–6 demoram o tempo $O(1)$. Como $m \leq n - 1$, o laço **for** das linhas 7–10 é executado no máximo $n - 3$ vezes. Visto que PUSH demora o tempo $O(1)$, cada iteração demora o tempo $O(1)$ excluído o tempo gasto no laço **while** das linhas 8–9; assim, o tempo global para laço **for** é $O(n)$, excluído o laço **while** aninhado.

Usamos análise agregada para mostrar que o laço **while** demora o tempo global $O(n)$. Para $i = 0, 1, ..., m$, empurramos cada ponto p_i para a pilha S exatamente uma vez. Como na análise do procedimento MULTIPOP da Seção 17.1, observamos que podemos extrair no máximo o mesmo número de itens que inserimos. No mínimo três pontos — p_0, p_1 e p_m — nunca são extraídos da pilha, de modo que, na verdade, no máximo $m - 2$ operações POP são executadas no total. Cada iteração do laço **while** executa uma operação POP e, portanto, há no máximo $m - 2$ iterações do laço **while** no total. Visto que o teste na linha 8 demora o tempo $O(1)$, cada chamada de POP demora o tempo $O(1)$, e $m \leq n - 1$, o tempo total para a execução do laço **while**, é $O(n)$. Assim, o tempo de execução de GRAHAM-SCAN é $O(n \lg n)$.

Marcha de Jarvis

A ***marcha de Jarvis*** calcula a envoltória convexa de um conjunto Q de pontos por uma técnica conhecida como ***embrulhar pacote*** (ou ***embrulhar presente***). O algoritmo é executado no tempo $O(nh)$, onde h é o número de vértices de CH(Q). Quando h é $o(\lg n)$, a marcha de Jarvis é assintoticamente mais rápida que a varredura de Graham.

Intuitivamente, a marcha de Jarvis simula a ação de embrulhar o conjunto Q com uma folha de papel bem esticada. Começamos colocando a extremidade do papel no ponto mais baixo do conjunto, isto é, no mesmo ponto p_0 com o qual iniciamos a varredura de Graham. Sabemos que esse ponto deve ser um vértice da envoltória convexa. Puxamos o papel para a direita para esticá-lo e depois para cima até tocar um ponto. Esse ponto também deve ser um vértice da envoltória convexa. Mantendo o papel esticado, continuamos desse modo ao redor do conjunto de vértices até voltarmos ao nosso ponto original p_0.

Em termos mais formais, a marcha de Jarvis constrói uma sequência $H = \langle p_1, p_2, ..., p_{h-1} \rangle$ dos vértices de CH(Q). Começamos com p_0. Como mostra a Figura 33.9, o próximo vértice p_1 da envoltória convexa tem o menor ângulo polar em relação a p_0. (No caso de empates, escolhemos o ponto mais afastado de p_0.) De modo semelhante, p_2 tem o menor ângulo polar em relação a p_1, e assim por diante. Quando chegamos ao vértice mais alto, digamos p_k (resolvendo empates escolhendo o vértice mais alto, mais afastado), construímos, como mostra a Figura 33.9, a ***cadeia da direita*** de CH(Q). Para construir a ***cadeia da esquerda***, começamos em p_k e escolhemos p_k+1 como o ponto que tem o menor ângulo polar em relação a p_k, mas *a partir do eixo x negativo*. Continuamos assim, formando a cadeia da esquerda, tomando ângulos polares em relação ao eixo x negativo até voltarmos ao nosso vértice original p_0.

Poderíamos implementar a marcha de Jarvis em uma única varredura conceitual em torno da envoltória convexa, isto é, sem construir separadamente as cadeias da direita e da esquerda. Em geral, tais implementações controlam o ângulo do último lado da envoltória convexa escolhido e exigem que a sequência de ângulos de lados da envoltória seja estritamente crescente (no intervalo de 0 a 2p radianos). A vantagem de construir cadeias separadas é que não precisamos calcular ângulos explicitamente; as técnicas da Seção 33.1 são suficientes para comparar ângulos.

Se implementada corretamente, o tempo de execução da marcha de Jarvis é $O(nh)$. Para cada um dos h vértices de CH(Q), determinamos o vértice que tem o menor ângulo polar. Cada comparação entre ângulos polares demora o tempo $O(1)$, usando as técnicas da Seção 33.1. Como mostra a Seção 9.1, podemos calcular o mínimo de n valores no tempo $O(n)$ se cada comparação demorar o tempo $O(1)$. Assim, a marcha de Jarvis demora o tempo $O(nh)$.

Exercícios

33.3-1 Prove que, no procedimento GRAHAM-SCAN, os pontos p_1 e p_m devem ser vértices de CH(Q).

33.3-2 Considere um modelo de computação que suporte adição, comparação e multiplicação, e para o qual exista um limite inferior (n lg n) para ordenar n números. Prove que $\Omega(n \lg n)$ é um limite inferior para calcular, em ordem, os vértices da envoltória convexa de um conjunto de n pontos em tal modelo.

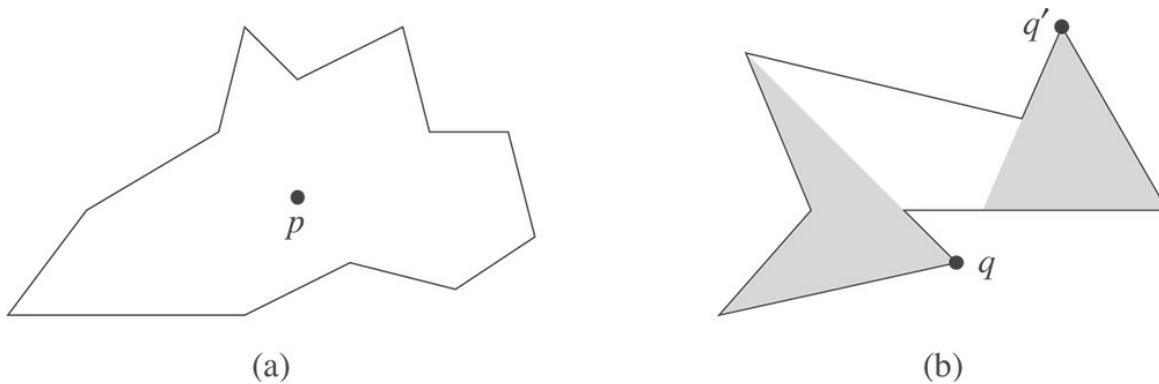
33.3-3 Dado um conjunto de pontos Q, prove que o par de pontos mais afastados entre si deve ser de vértices de CH(Q).

33.3-4 Para um dado polígono P e um ponto q em seu contorno, a ***sombra*** de q é o conjunto de pontos r tal que o segmento qr está inteiramente sobre o contorno ou no interior de P. Como ilustra a Figura 33.10, um polígono P tem ***formato de estrela*** (ou é ***estrelado***) se existe um ponto p no interior de P que esteja na sombra de todo ponto no contorno de P. O conjunto de todos esses pontos p é denominado ***núcleo*** de P. Dado um polígono estrelado P de n vértices especificado por seus vértices em ordem anti-horária, mostre como calcular CH(P) no tempo $O(n)$.

Figura 33.9 A operação da marcha de Jarvis. O primeiro vértice escolhido é o ponto mais baixo p_0. O vértice seguinte, p_1, tem o menor ângulo polar de qualquer ponto em relação a p_0. Então, p_2 tem o menor ângulo polar em relação a p_1. A cadeia da direita vai até o ponto mais alto p_3. Então, construímos a cadeia da esquerda determinando os menores ângulos polares em relação ao eixo x negativo.

33.3-5 No ***problema da envoltória convexa on-line***, temos o conjunto Q de n pontos, um ponto por vez. Depois de receber cada ponto, calculamos a envoltória convexa dos pontos vistos até o momento. É óbvio que poderíamos executar a varredura de Graham uma vez para cada ponto, com tempo total de execução $O(n_2$ lg

n). Mostre como resolver o problema da envoltória convexa on-line no tempo total $O(n_2)$.

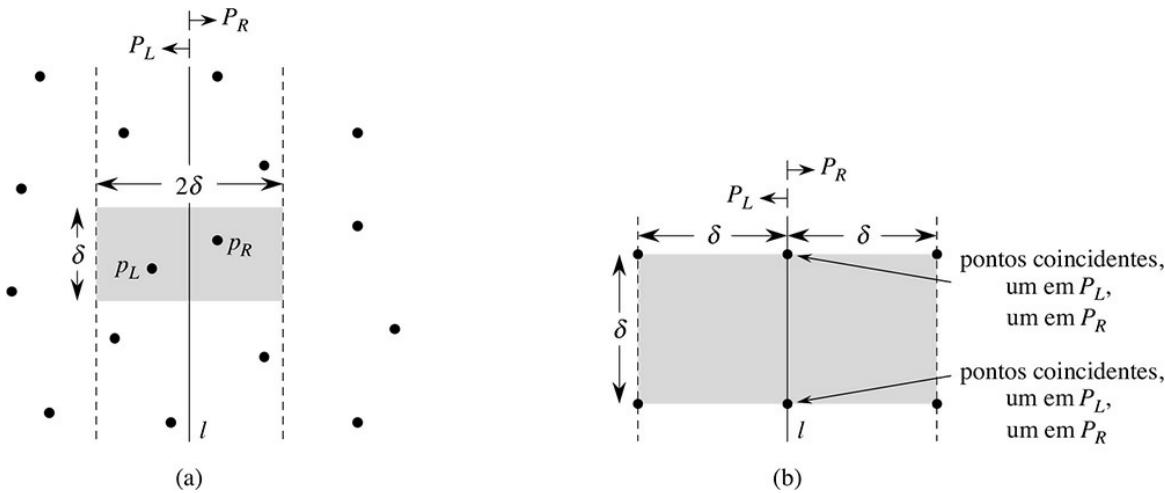
33.3-6 ★ Mostre como implementar o método incremental para calcular a envoltória convexa de n pontos, de modo que ele seja executado no tempo $O(n \lg n)$.

33.4 LOCALIZANDO O PAR DE PONTOS MAIS PRÓXIMOS

Agora, consideramos o problema de determinar o par de pontos mais próximos em um conjunto Q de $n \geq 2$ pontos. A expressão "mais próximos" se refere à distância euclidiana habitual: a distância entre os pontos $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ é $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Dois pontos no conjunto Q podem ser coincidentes e, nesse caso, a distância entre eles é zero. Esse problema tem aplicação, por exemplo, em sistemas de controle de tráfego. Um sistema de controle de tráfego aéreo ou marítimo deve identificar os dois veículos que estão mais próximos um do outro para detectar colisões potenciais.

Um algoritmo de força bruta para pares mais próximos simplesmente examina todos os pares $\binom{n}{2} = \Theta(n^2)$ de pontos. Nesta seção, descreveremos um algoritmo de divisão e conquista para esse problema, cujo tempo de execução é descrito pela conhecida recorrência $T(n) = 2T(n/2) + O(n)$. Assim, o algoritmo usa somente o tempo $O(n \lg n)$.

O algoritmo de divisão e conquista

Cada chamada recursiva do algoritmo toma como entrada um subconjunto $P \subseteq Q$ e os arranjos X e Y, cada um contendo todos os pontos do subconjunto de entrada P. Os pontos no arranjo X são ordenados de modo que suas coordenadas x são monotonicamente crescentes. De modo semelhante, o arranjo Y é ordenado por coordenada y monotonicamente crescente. Observe que, para conseguir o limite de tempo $O(n \lg n)$ não podemos nos dar ao luxo de efetuar ordenação em cada chamada recursiva; se o fizéssemos, a recorrência para o tempo de execução seria $T(n) = 2T(n/2) + O(n \lg n)$, cuja solução é $T(n) = O(n \lg^2 n)$. (Use a versão do método mestre dada no Exercício 4.6-2.) Veremos um pouco mais adiante como usar "pré-ordenação" para manter essa propriedade ordenada sem realmente efetuar ordenação em cada chamada recursiva.

Figura 33.10 A definição de um polígono estrelado para uso no Exercício 33.3-4. (**a**) Um polígono estrelado. O segmento do ponto p até qualquer ponto q no contorno intercepta o contorno somente em q. (**b**) Um polígono não estrelado. A região sombreada à esquerda é a sombra de q, e a região sombreada à direita é a sombra de q'. Como essas regiões são disjuntas, o núcleo é vazio.

Uma dada invocação recursiva com entradas P, X e Y , primeiro verifica se $|P| \leq 3$. Se for, a invocação simplesmente executa o método de força bruta já descrito: tentar todos os pares de pontos e retornar o par mais próximo. Se $|P| > 3$, a invocação recursiva executa o paradigma de divisão e conquista descrito a seguir.

Divisão: Determine uma reta vertical l que divide o conjunto de pontos P em dois conjuntos P_L e P_R tais que $|P_L| = | P|/2$, $| P_R| = | P|/2$, todos os pontos em P_L estão sobre ou à esquerda da reta l e todos os pontos em P_R estão sobre ou à esquerda de l. Divida o arranjo X em arranjos X_L e X_R, que contêm os pontos de P_L e P_R, respectivamente, ordenados por coordenada x monotonicamente crescente. De modo semelhante, divida o arranjo Y nos arranjos Y_L e Y_R, que contêm os pontos de P_L e P_R, respectivamente, ordenados por coordenada y monotonicamente crescente.

Conquista: Agora, que temos P dividido em P_L e P_R, faça duas chamadas recursivas, uma para encontrar o par de pontos mais próximos em P_L e a outra para encontrar o par de pontos mais próximos em P_R. As entradas para a primeira chamada são o subconjunto P_L e arranjos X_L e Y_L; a segunda chamada recebe as entradas P_R, X_R e Y_R. Sejam d_L e d_R as distâncias de pares mais próximos retornadas para P_L e P_R, respectivamente, e seja $d = \min(d_L, d_R)$.

Combinar: O par de pontos mais próximos é o par com distância d encontrado por uma das chamadas recursivas ou é o par de pontos com um ponto em P_L e o outro em P_R. O algoritmo determina se existe um par com um ponto em P_L e o outro ponto em P_R cuja distância é menor que d. Observe que, se existe um par de pontos com distância menor que d, ambos os pontos do par devem estar a d unidades em relação à reta l. Assim, como mostra a Figura 33.11(a), ambos devem estar na faixa vertical de largura $2d$ com centro na reta l. Para encontrar tal par, se existir algum, o algoritmo faz o seguinte:

1. Cria um arranjo Y', que é o arranjo Y do qual todos os pontos que não estão na faixa vertical de largura $2d$ foram eliminados. O arranjo Y' é ordenado por coordenada y exatamente como Y.

Figura 33.11 Conceitos fundamentais na prova de que o algoritmo de pares de pontos mais próximos precisa verificar somente sete pontos que vêm depois de cada ponto no arranjo Y'. (**a**) Se $p_L \in P_L$ e $p_R \in P_R$ estão a menos de d unidades um do outro, eles devem estar dentro de um retângulo $d \times 2d$ com centro na reta l. (**b**) Como quatro pontos que, aos pares, estão no mínimo a d unidades um do outro, podem estar dentro de um quadrado $d \times d$. À esquerda estão quatro pontos em P_L e à direita estão quatro pontos em P_R. O retângulo $d \times 2d$ pode conter oito pontos se os pontos mostrados sobre a reta l são, na realidade, pares de pontos coincidentes com um ponto em P_L e um em P_R.

2. Para cada ponto p no arranjo Y', o algoritmo tenta encontrar pontos em Y' que estejam a d unidades em relação a p. Como veremos em breve, apenas os sete pontos em Y' que vêm depois de p precisam ser considerados. O

algoritmo calcula a distância de p até cada um desses sete pontos e controla a distância d' do par de pontos mais próximos encontrada para todos os pares de pontos em Y'.

3. Se $d' < d$, então a faixa vertical de fato contém um par mais próximo do que aquele encontrado pelas chamadas recursivas. O algoritmo retorna esse par e sua distância d'. Caso contrário, o algoritmo retorna o par de pontos mais próximos e sua distância d encontrados pelas chamadas recursivas.

Essa descrição omite alguns detalhes de implementação que são necessários para conseguir o tempo de execução $O(n \lg n)$. Depois de provar a correção do algoritmo, mostraremos como implementá-lo de modo a alcançar o limite de tempo desejado.

Correção

A correção desse algoritmo de pares mais próximos é óbvia, exceto por dois aspectos. O primeiro é que, interrompendo a recursão quando $|P| \leq 3$, garantimos que nunca tentaremos resolver um subproblema que consista em apenas um ponto. O segundo é que precisamos verificar somente os sete pontos que vêm depois de cada ponto p no arranjo Y'. Agora, provaremos essa propriedade.

Suponha que, em algum nível da recursão, o par de pontos mais próximos seja $p_L \in P_L$ e $p_R \in P_R$. Assim, a distância d' entre p_L e p_R é estritamente menor que d. O ponto p_L deve estar sobre ou à esquerda da reta l e a menos de d unidades em relação a essa reta. De modo semelhante, p_R está sobre ou à direita de l e a menos de d unidades em relação a essa reta. Além disso, p_L e p_R estão a d unidades um do outro na direção vertical. Assim, como mostra a Figura 33.11(a), p_L e p_R estão dentro de um retângulo $d \times 2d$ com centro na reta l. (Também é possível que existam outros pontos dentro desse retângulo.)

Mostraremos em seguida que no máximo oito pontos de P podem residir dentro desse retângulo $d \times 2d$. Considere o quadrado $d \times d$ que forma a metade esquerda desse retângulo. Visto que todos os pontos dentro de P_L estão no mínimo a d unidades um do outro, no máximo quatro pontos podem residir dentro desse quadrado; a Figura 33.11(b) mostra como. De modo semelhante, no máximo quatro pontos em P_R podem residir dentro do quadrado $d \times d$ que forma a metade direita do retângulo. Assim, no máximo oito pontos de P podem residir no interior do retângulo $d \times 2d$. (Observe que, como os pontos sobre a reta l podem estar em P_L ou em P_R, pode haver até quatro pontos sobre l. Esse limite é alcançado se existem dois pares de pontos coincidentes, tais que cada par consista em um ponto de P_L e um ponto de P_R, um par esteja na interseção de l com a parte superior do retângulo e o outro par no local em que l intercepta a parte inferior do retângulo.)

Agora, que já mostramos que no máximo oito pontos de P podem residir no interior do retângulo, é fácil ver que precisamos verificar somente os sete pontos que vêm depois de cada ponto no arranjo Y'. Ainda considerando que o par de pontos mais próximos seja p_L e p_R, vamos supor, sem perda da generalidade, que p_L preceda p_R no arranjo Y'. Então, mesmo que p_L ocorra o mais cedo possível em Y' e p_R ocorra o mais tarde possível, p_R está em uma das sete posições que vêm depois de p_L. Portanto, mostramos a correção do algoritmo de pares mais próximos.

Implementação e tempo de execução

Como observamos, nossa meta é conseguir que a recorrência para o tempo de execução seja $T(n) = 2T(n/2) + O(n)$, onde $T(n)$ é o tempo de execução para um conjunto de n pontos. A principal dificuldade está em assegurar que os arranjos X_L, X_R, Y_L e Y_R, que são repassados para as chamadas recursivas, sejam ordenados pela coordenada adequada e também que o arranjo Y' seja ordenado pela coordenada y. (Observe que, se o arranjo X que é recebido por uma chamada recursiva já estiver ordenado, é fácil dividir o conjunto P em P_L e P_R em tempo linear.)

A observação fundamental é que, em cada chamada, desejamos formar um subconjunto ordenado de um arranjo ordenado. Por exemplo, uma invocação particular recebe o subconjunto P e o arranjo Y, ordenado pela coordenada y. Como foi particionado em P_L e P_R, P precisa formar os arranjos Y_L e Y_R, que são ordenados pela coordenada y em

tempo linear. Podemos ver o método como o oposto do procedimento MERGE da ordenação por intercalação na Seção 2.3.1: estamos dividindo um arranjo ordenado em dois arranjos ordenados. O pseudocódigo a seguir dá a ideia.

```
1 sejam Y_L[1. . Y. comprimento] e Y_R[1. . Y. comprimento] novos arranjos
2 Y_L. comprimento = Y_R. comprimento = 0
3 for i = 1 to Y. comprimento
4     if Y [i] ∈ P_L
5         Y_L. comprimento = Y_L. comprimento + 1
6         Y_L[Y_L. comprimento] = Y [i]
7     else Y_R. comprimento = Y_R. comprimento + 1
8         Y_R[Y_R. comprimento] = Y [i]
```

Simplesmente examinamos os pontos no arranjo Y em ordem. Se um ponto $Y[i]$ está em P_L, anexamos o ponto ao final do arranjo Y_L; caso contrário, nós o anexamos ao final do arranjo Y_R. Um pseudocódigo semelhante funciona para formar os arranjos X_L, X_R e Y'.

A única pergunta que resta é como obter os pontos ordenados, antes de mais nada. Nós os ***pré-ordenamos***; isto é, ordenamos os pontos de uma vez por todas *antes* da primeira chamada recursiva. Passamos esses arranjos ordenados para a primeira chamada recursiva e, daí em diante, os reduzimos gradualmente por meio das chamadas recursivas necessárias. A pré-ordenação acrescenta um termo $O(n \lg n)$ adicional ao tempo de execução, mas agora cada etapa da recursão demora tempo linear, excluídas as chamadas recursivas. Assim, se $T(n)$ é o tempo de execução de cada etapa recursiva e $T'(n)$ é o tempo de execução do algoritmo inteiro, obtemos

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{se } n > 3, \\ O(1) & \text{se } n \leq 3. \end{cases}$$

Assim, $T(n) = O(n \lg n)$ e $T'(n) = O(n \lg n)$.

Exercícios

33.4-1 O professor Williams propõe um esquema que permite que o algoritmo de pares mais próximos verifique somente cinco pontos que vêm depois de cada ponto no arranjo Y'. A ideia é sempre colocar pontos sobre a reta l no conjunto P_L. Então, não pode haver pares de pontos coincidentes sobre a reta l com um ponto em P_L e um em P_R. Assim, no máximo seis pontos podem residir no retângulo $d \times 2d$. Qual é a falha no esquema do professor?

33.4-2 Mostre que, na verdade, basta verificar somente os pontos nas cinco posições do arranjo que vêm depois de cada ponto no arranjo Y′.

33.4-3 Podemos definir a distância entre dois pontos de outras modos, além do euclidiano. No plano, a ***distância*** $\boldsymbol{L_m}$ entre os pontos p_1 e p_2 é dada pela expressão $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$. Portanto, a distância euclidiana é distância L_2. Modifique o algoritmo de pares mais próximos para usar a distância L_1, que também é conhecida como ***distância Manhattan***.

33.4-4 Dados dois pontos p_1 e p_2 no plano, a distância L entre eles é dada por max $(|x_1 - x_2|, |y_1 - y_2|)$. Modifique o algoritmo do par mais próximo para utilizar a distância L.

33.4-5 Suponha que $\Omega(n)$ dos pontos dados ao algoritmo do par mais próximo são coverticais. Mostre como determinar os conjuntos P_L e P_R e como determinar se cada ponto de Y está em P_L ou P_R de modo tal que o tempo de execução para o algoritmo do par mais próximo permaneça $O(n \lg n)$.

33.4-6 Sugira uma mudança no algoritmo do par mais próximo que evite a pré-ordenação do arranjo Y, mas deixe o tempo de execução como $O(n \lg n)$. (*Sugestão:* Intercale arranjos ordenados Y_L e Y_R para formar o arranjo ordenado Y.)

Problemas

35-1 Camadas convexas

Dado um conjunto Q de pontos no plano, definimos as ***camadas convexas*** de Q por indução. A primeira camada convexa de Q consiste nos pontos em Q que são vértices de CH(Q). Para $i > 1$, definimos Q_i de modo que consista nos pontos em Q, eliminados todos os pontos em camadas convexas 1, 2, ..., $i - 1$. Então, a i-ésima camada convexa de Q é CH(Q_i) se $Q_i \neq /0$ e é indefinida, caso contrário.

a. Dê um algoritmo de tempo $O(n^2)$ para encontrar as camadas convexas de um conjunto de n pontos.

b. Prove que é necessário o tempo $\Omega(n \lg n)$ para calcular as camadas convexas de um conjunto de n pontos com qualquer modelo de computação que exija o tempo $(n \lg n)$ para ordenar n números reais.

35-2 Camadas máximas

Seja Q um conjunto de n pontos no plano. Dizemos que o ponto (x, y) ***domina*** o ponto (x', y') se $x \geq x'$ e $y \geq y'$. Um ponto em Q que não é dominado por nenhum outro ponto em Q é denominado ***maximal***. Observe que Q pode conter muitos pontos máximos, que podem ser organizados em ***camadas maximais*** da seguinte maneira. A primeira camada maximal L_1 é o conjunto de pontos máximais em Q. Para $i > 1$, a i-ésima camada maximal i é o conjunto de pontos maximais em $Q - \bigcup_{j=1}^{i-1} Lj$.

Suponha que Q tenha k camadas máximas não vazias e seja y_i a coordenada y do ponto mais à esquerda em L_i para $i = 1, 2, ..., k$. Por enquanto, supomos que não haja dois pontos em Q com a mesma coordenada x ou y.

a. Mostre que $y_1 > y_2 > ... > y_k$.

Considere um ponto (x, y) que esteja à esquerda de qualquer ponto em Q e para o qual y seja distinto da coordenada y de qualquer ponto em Q. Seja $Q' = Q \cup \{(x, y)\}$.

b. Seja j o índice mínimo tal que $y_j < y$, a menos que $y < y_k$, caso em que fazemos $j = k + 1$. Mostre que as camadas maximais de Q' são as seguintes.

- Se $j \leq k$, então as camadas maximais de Q' são as mesmas que as camadas maximais de Q, exceto que L_j também inclui (x, y) como seu novo ponto mais à esquerda.
- Se $j = k + 1$, então as primeiras k camadas máximas de Q' são as mesmas que as de Q mas, além disso, Q' tem uma $(k + 1)$-ésima camada máxima não vazia: $L_{k+1} = \{(x, y)\}$.

c. Descreva um algoritmo de tempo $O(n \lg n)$ para calcular as camadas maximais de um conjunto Q de n pontos. (*Sugestão*: Movimente uma linha de varredura da direita para a esquerda.)

d. Surge alguma dificuldade se agora permitirmos que pontos de entrada tenham a mesma coordenada x ou y? Sugira um modo de solucionar tais problemas.

35-3 Caça-fantasmas e fantasmas

Um grupo de *n* caça-fantasmas está perseguindo *n* fantasmas. Cada caça-fantasmas está armado com um pacote de prótons, que dispara um feixe em um fantasma, erradicando-o. Um feixe segue em linha reta e termina quando atinge o fantasma. Os caça-fantasmas decidem adotar a estratégia a seguir. Eles formarão pares com os fantasmas, constituindo *n* pares caça-fantasma e então, simultaneamente, cada caça-fantasmas dispara um feixe no fantasma que escolheu. Como sabemos, é *muito* perigoso permitir que os feixes se cruzem e, assim, os caça-fantasmas devem escolher pares de modo que nenhum feixe cruze com outro.

Suponha que a posição de cada caça-fantasmas e cada fantasma sejaum ponto fixo no plano e que não haja três posições que sejam colineares.

a. Demonstre que existe uma reta que passa por um caça-fantasmas e um fantasma tal que o número de caça-fantasmas em um lado da reta seja igual ao número de fantasmas no mesmo lado. Descreva como determinar tal reta no tempo $O(n \lg n)$.

b. Dê um algoritmo de tempo $O(n_2 \lg n)$ para formar os pares caça-fantasmas/fantasmas de tal modo que nenhum feixe cruze outro.

35-4 Pega-varetas

O professor Charon tem um conjunto de *n* varetas, empilhadas umas sobre as outras em alguma configuração. Cada vareta é especificada por suas extremidades e cada extremidade é uma tripla ordenada que dá suas coordenadas (x, y, z). Nenhuma vareta está na vertical. Ele deseja retirar todas as varetas, uma de cada vez, sob a condição de que ele só pode retirar uma vareta se não houver outra vareta sobre ela.

a. Dê um procedimento que toma duas varetas *a* e *b* e informa se *a* está acima, abaixo ou não está relacionada com *b*.

b. Descreva um algoritmo eficiente que determine se é possível retirar todas as varetas e, se isso for possível, dê uma ordem válida para a retirada das varetas.

35-5 Distribuições de envoltórias esparsas

Considere o problema de calcular a envoltória convexa de um conjunto de pontos no plano que tenham sido extraídos de acordo com alguma distribuição aleatória conhecida. Às vezes, o número de pontos, ou tamanho, da envoltória convexa de *n* pontos extraídos de tal distribuição tem esperança $O(n_1$-) para alguma constante > 0. Dizemos que tal distribuição é de ***envoltória esparsa***. As distribuições de envoltória esparsa incluem o seguinte:

- Pontos extraídos uniformemente de um disco de raio unitário. A envoltória convexa tem tamanho esperado Q($n_{1/3}$).
- Pontos extraídos uniformemente do interior de um polígono convexo com *k* lados, para qualquer constante *k*. A envoltória convexa tem tamanho esperado Q(lg *n*).
- Pontos extraídos de acordo com uma distribuição normal bidimensional. A envoltória convexa tem o tamanho esperado Q($\sqrt{Ig\ n}$).

a. Dados dois polígonos convexos com n_1 e n_2 vértices, respectivamente, mostre como calcular a envoltória convexa de todos os $n_1 + n_2$ pontos no tempo $O(n_1 + n_2)$. (Os polígonos podem se sobrepor.)

b. Mostre como calcular a envoltória convexa de um conjunto de n pontos extraídos independentemente de acordo com uma distribuição de envoltórias esparsas no tempo do caso médio $O(n)$. (*Sugestão*: Determine recursivamente as envoltórias convexas dos primeiros $n/2$ pontos e dos $n/2$ pontos seguintes e depois combine os resultados.)

NOTAS DO CAPÍTULO

Este capítulo mal arranha a superfície dos algoritmos e técnicas de geometria computacional. Livros sobre geometria computacional incluem os de Preparata e Shamos [282], Edelsbrunner [99] e O'Rourke [269].

Embora a geometria tenha sido estudada desde a Antiguidade, o desenvolvimento de algoritmos para problemas geométricos é relativamente novo. Preparata e Shamos observam que a primeira noção da complexidade de um problema foi dada por E. Lemoine em 1902. Ele estava estudando construções euclidianas — aquelas que usam uma régua e um compasso — e criou um conjunto de cinco primitivas: colocar uma perna do compasso sobre um ponto dado, colocar uma perna do compasso sobre uma reta dada, desenhar um círculo, passar a borda da régua por um ponto dado e desenhar uma reta. Lemoine estava interessado no número de primitivas necessárias para executar uma determinada construção; ele denominou essa quantidade "simplicidade" da construção.

O algoritmo da Seção 33.2, que determina se quaisquer segmentos se interceptam, se deve a Shamos e Hoey [313].

A versão original da varredura de Graham é dada por Graham [150]. O algoritmo de embrulhar pacote se deve a Jarvis [189]. Usando um modelo de computação de árvore de decisão, Yao [359] provou o limite inferior de $\Omega(n \lg n)$ para o tempo de execução de qualquer algoritmo de envoltórias convexas. Quando o número de vértices h da envoltória convexa é levado em conta, o algoritmo de poda e busca de Kirkpatrick e Seidel [206], que demora o tempo $O(n \lg h)$, é assintoticamente ótimo.

O algoritmo de divisão e conquista de tempo $O(n \lg n)$ para determinar o par de pontos mais próximo foi criado por Shamos e aparece em Preparata e Shamos [282]. Preparata e Shamos também mostram que o algoritmo é assintoticamente ótimo em um modelo de árvore de decisão.

---

[1] Na realidade, o produto cruzado é um conceito tridimensional. É um vetor perpendicular a p_1 e p_2, de acordo com a "regra da mão direita" e cuja magnitude é $|x_1y_2 - x_2 y_1|$. Contudo, neste capítulo, achamos conveniente tratar o produto cruzado apenas como o valor $x_1y_2 - x_2 y_1$.

[2] Se permitirmos que três segmentos se interceptem no mesmo ponto, pode haver um segmento c interveniente que intercepta a e b no ponto p. Isto é, podemos ter $a <_w c$ e $c <_w b$ para todas as linhas de varredura w à esquerda de p para as quais $a <_w b$. O Exercício 33.2-8 pede que você mostre que Any-Segments-Intersect é correto ainda que três segmentos se interceptem no mesmo ponto.

34 PROBLEMAS NP-COMPLETOS

Quase todos os algoritmos que estudamos até aqui são ***algoritmos de tempo polinomial***: para entradas de tamanho n, seu tempo de execução do pior caso é $O(n_k)$ para alguma constante k. É natural imaginar que *todos* os problemas podem ser resolvidos em tempo polinomial. A resposta é não. Por exemplo, existem problemas, como o famoso "problema da parada" de Turing, que não podem ser resolvidos por qualquer computador, não importa por quanto tempo seja executado. Também existem problemas que podem ser resolvidos, mas não no tempo $O(n_k)$ para qualquer constante k. Em geral, pensamos que problemas que podem ser resolvidos por algoritmo de tempo polinomial são tratáveis, ou fáceis, e que problemas que exigem tempo superpolinomial são intratáveis ou difíceis.

Porém, o assunto deste capítulo é uma classe interessante de problemas, denominados problemas "NP-completos", cujo status é desconhecido. Ainda não foi descoberto nenhum algoritmo de tempo polinomial para um problema NP-completo e ninguém ainda conseguiu provar que não pode existir nenhum algoritmo de tempo polinomial para nenhum deles. Essa questão denominada P ≠ NP continua sendo um dos mais profundos e intrigantes problemas de pesquisa ainda em aberto na teoria da ciência da computação, desde que foi proposto pela primeira vez em 1971.

Vários problemas NP-completos são particularmente torturantes porque, à primeira vista, parecem ser semelhantes a problemas que sabemos resolver em tempo polinomial. Em cada um dos pares de problemas a seguir, um deles pode ser resolvido em tempo polinomial e o outro é NP-completo, mas a diferença entre eles parece insignificante:

Caminhos simples mínimos e caminhos simples de comprimento máximo: No Capítulo 24, vimos que até mesmo com pesos de arestas negativos podemos encontrar caminhos mínimos que partem de uma única origem em um grafo dirigido $G = (V, E)$ no tempo $O(VE)$. Contudo, determinar o caminho simples *de comprimento máximo* entre dois vértices é difícil. Simplesmente determinar se um grafo contém um caminho simples com pelo menos um determinado número de arestas é NP-completo.

Passeio de Euler e ciclo hamiltoniano: Um ***passeio de Euler*** de um grafo conexo dirigido $G = (V, E)$ é um ciclo que percorre cada *aresta* de G exatamente uma vez, embora possa visitar cada vértice mais de uma vez. Pelo Problema 22-3, podemos determinar se um grafo tem um passeio de Euler no tempo de apenas $O(E)$ e, de fato, podemos encontrar as arestas do passeio de Euler no tempo $O(E)$. Um ***ciclo hamiltoniano*** de um grafo dirigido $G = (V, E)$ é um ciclo simples que contém cada *vértice* em V. Determinar se um grafo dirigido tem um ciclo hamiltoniano é NP-completo. (Mais adiante neste capítulo provaremos que determinar se um grafo *não dirigido* tem um ciclo hamiltoniano é NP-completo.)

Satisfazibilidade 2-CNF e satisfazibilidade 3-CNF: Uma fórmula booleana contém variáveis cujos valores são 0 ou 1; conectivos booleanos como $\wedge$ (AND), $\vee$ (OR) e $\neg$ (NOT); e parênteses. Uma fórmula booleana é ***satisfazível*** se existe alguma atribuição dos valores 0 e 1 às suas variáveis que faça com que ela seja avaliada como 1. Definiremos os termos em linguagem mais formal mais adiante neste capítulo; porém, informalmente, uma fórmula booleana está em ***forma normal k-conjuntiva***, ou k-CNF, se for o AND de cláusulas OR de exatamente k variáveis ou de suas negações. Por exemplo, a fórmula booleana $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3)$ está em 2-CNF. (Ela tem a atribuição que satisfaz $x_1 = 1$, $x_2 = 0$, $x_3 = 1$.) Embora possamos

determinar em tempo polinomial s uma fórmula 2-CNF é satisfazível, veremos mais adiante neste capítulo que determinar se uma fórmula 3-CNF é satisfazível é NP-completo.

NP-completude e as classes P e NP

Ao longo deste capítulo, nos referiremos a três classes de problemas: P, NP e NPC, sendo a última classe a dos problemas NP-completos. Aqui, as descrevemos de um modo informal; mais adiante, as definiremos em linguagem mais formal.

A classe P consiste nos problemas que podem ser resolvidos em tempo polinomial. Mais especificamente, são problemas que podem ser resolvidos no tempo $O(n_k)$ para alguma constante k, onde n é o tamanho da entrada para o problema. A maioria dos problemas examinados em capítulos anteriores pertence à classe P.

A classe NP consiste nos problemas que são "verificáveis" em tempo polinomial. O que significa um problema ser verificável? Se tivéssemos algum tipo de "certificado" de uma solução, poderíamos verificar se o certificado é correto em tempo polinomial para o tamanho da entrada para o problema. Por exemplo, no problema do ciclo hamiltoniano, dado um grafo dirigido $G = (V, E)$, um certificado seria uma sequência $\langle v_1, v_2, v_3, \cdots, v|V| \rangle$ de $|V|$ vértices. É fácil verificar em tempo polinomial que $(v_i, v_i{+}1) \in E$ para $i = 1, 2, 3, ..., |V| - 1$ e também que $(v|V|, v_1) \in E$. Como outro exemplo; para satisfazibilidade 3-CNF um certificado seria uma atribuição de valores a variáveis. Poderíamos verificar em tempo polinomial que essa atribuição satisfaz a fórmula booleana.

Qualquer problema em P também está em NP visto que, se um problema está em P, podemos resolvê-lo em tempo polinomial sem nem mesmo ter um certificado. Formalizaremos essa noção mais adiante neste capítulo mas, por enquanto, podemos acreditar que P ⊆ NP. A questão em aberto é se P é ou não um subconjunto próprio de NP.

Informalmente, um problema está na classe NPC — e nos referiremos a ele como um problema ***NP-completo*** — se ele está em NP e é tão "difícil" quanto qualquer problema em NP. Definiremos formalmente o que significa ser tão difícil quanto qualquer problema em NP mais adiante neste capítulo. Enquanto isso, afirmaremos sem provar que, se *qualquer* problema NP-completo pode ser resolvido em tempo polinomial, então *todo* problema em NP tem um algoritmo de tempo polinomial. A maioria dos teóricos da ciência da computação acredita que os problemas NP-completos sejam intratáveis já que, dada a ampla faixa de problemas NP-completos que foram estudados até hoje — sem que ninguém tenha descoberto uma solução de tempo polinomial para nenhum deles — seria verdadeiramente espantoso se todos eles pudessem ser resolvidos em tempo polinomial. Ainda assim, dado o esforço dedicado até agora para provar que os problemas NP-completos são intratáveis — sem um resultado conclusivo — não podemos descartar a possibilidade de que os problemas NP-completos são de fato resolvíveis em tempo polinomial.

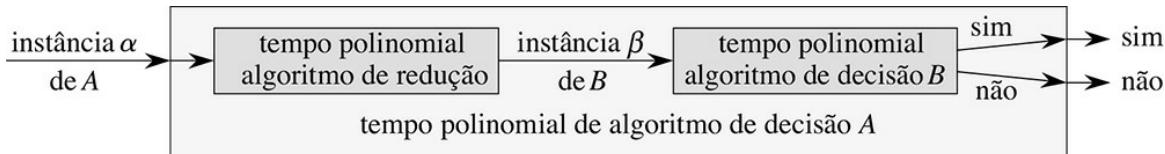
Para se tornar um bom projetista de algoritmos, você deve entender os rudimentos da teoria da NP-completude. Se puder determinar que um problema é NP-completo, estará dando uma boa evidência de sua intratabilidade. Então, como engenheiro, seria melhor empregar seu tempo no desenvolvimento de um algoritmo de aproximação (veja o Capítulo 35) ou resolvendo um caso especial tratável, em vez de procurar um algoritmo rápido que resolva o problema exatamente. Além disso, muitos problemas naturais e interessantes que à primeira vista não parecem mais difíceis que ordenação, busca de grafos ou fluxo em rede são de fato NP-completos. Portanto, é importante se familiarizar com essa classe notável de problemas.

Como mostrar que um problema é NP-completo: uma visão geral

As técnicas que empregamos para mostrar que um determinado problema é NP-completo são fundamentalmente diferentes das técnicas usadas em quase todo este livro para projetar e analisar algoritmos. Quando dizemos que um problema é NP-completo, fica subentendido que trata-se de um problema difícil de resolver (ou ao menos que achamos que é difícil), e não de um problema fácil de resolver. Não estamos tentando provar a existência de um algoritmo eficiente, mas que provavelmente não existe nenhum algoritmo eficiente. Por essa perspectiva, as provas da NP-completude são um pouco parecidas com a prova na Seção 8.1 de um limite inferior de tempo (n lg n) para qualquer

algoritmo de ordenação por comparação; contudo, as técnicas específicas usadas para mostrar a NP-completude são diferentes do método da árvore de decisão usado na Seção 8.1.

Contamos com três conceitos fundamentais para mostrar que um problema é NP-completo:

Problemas de decisão *versus* problemas de otimização

Muitos problemas de interesse são ***problemas de otimização***, para os quais cada solução possível (isto é, "válida") tem um valor associado e para os quais desejamos encontrar uma solução viável com o melhor valor. Por exemplo, em um problema que denominamos SHORTEST-PATH, temos um grafo não dirigido G e vértices u e v, e desejamos encontrar o caminho de u a v que utiliza o menor número de arestas. Em outras palavras, SHORTEST-PATH é o problema do caminho mínimo para um par em um grafo não ponderado e não dirigido). Porém, a NP-completude não se aplica diretamente a problemas de otimização, mas a ***problemas de decisão***, para os quais a resposta é simplesmente "sim" ou "não" (ou, em linguagem mais formal, "1" ou "0").

Embora problemas NP-completos estejam confinados ao reino dos problemas de decisão, podemos tirar proveito de uma relação conveniente entre problemas de otimização e problemas de decisão. Normalmente, podemos expressar um determinado problema de otimização como um problema de decisão relacionado impondo um limite para o valor a ser otimizado. Por exemplo, um problema de decisão relacionado com SHORTEST-PATH é PATH: dado um grafo dirigido G, vértices u e v, e um inteiro k, existe um caminho de u a v que consiste em no máximo k arestas?

A relação entre um problema de otimização e seu problema de decisão relacionado age a nosso favor quando tentamos mostrar que o problema de otimização é "difícil". Isso porque o problema de decisão é de certo modo "mais fácil" ou, ao menos, "não é mais difícil". Como um exemplo específico, podemos resolver PATH resolvendo SHORTEST-PATH e depois comparando o número de arestas no caminho mínimo encontrado com o valor do parâmetro k do problema de decisão. Em outras palavras, se um problema de otimização é fácil, seu problema de decisão relacionado também é fácil. Em termos mais relevantes para a NP-completude, se pudermos apresentar evidências de que um problema de decisão é difícil, também apresentamos evidências de que seu problema de otimização relacionado é difícil. Assim, embora restrinja a atenção a problemas de decisão, muitas vezes, a teoria da NP-completude também tem implicações para problemas de otimização.

Reduções

Essa ideia de mostrar que um problema não é mais difícil ou não é mais fácil que outro se aplica até mesmo quando ambos são problemas de decisão. Tiramos proveito dessa ideia em quase todas as provas da NP-completude, como veremos em seguida. Vamos considerar um problema de decisão A, que gostaríamos de resolver em tempo polinomial. Denominamos a entrada para um determinado problema por ***instância*** desse problema; por exemplo, em PATH, uma instância seria um grafo G dado, vértices específicos u e v de G e um determinado inteiro k. Agora, suponha que já sabemos como resolver um problema de decisão diferente, B, em tempo polinomial. Finalmente, suponha que temos um procedimento que transforma qualquer instância a de A em alguma instância b de B com as seguintes características:

Figura 34.1 Como usar um algoritmo de redução de tempo polinomial para resolver um problema de decisão A em tempo polinomial, dado um algoritmo de decisão de tempo polinomial para um outro problema B. Em tempo polinomial, transformamos uma instância a de A em uma instância b de B, resolvemos B em tempo polinomial e usamos a resposta para b como a resposta para α.

- A transformação demora tempo polinomial.
- As respostas são as mesmas. Isto é, a resposta para α é "sim" se e somente se a resposta para β também é "sim".

Denominamos tal procedimento ***algoritmo de redução*** de tempo polinomial e, como mostra a Figura 34.1, ele proporciona um meio para resolver o problema A em tempo polinomial:

1. Dada uma instância a do problema A, use um algoritmo de redução de tempo polinomial para transformá-la em uma instância b do problema B.
2. Execute o algoritmo de decisão de tempo polinomial para B para a instância b.
3. Use a resposta de b como a resposta para a.

Desde que cada uma dessas etapas demore tempo polinomial, as três juntas também demoram um tempo polinomial e, assim, temos um modo de decidir para α em tempo polinomial. Em outras palavras, "reduzindo" a solução do problema A à solução do problema B, usamos a "facilidade" de B para provar a "facilidade" de A.

Lembrando que a NP-completude consiste em mostrar o quanto um problema é difícil, em vez de mostrar o quanto ele é fácil, usamos reduções de tempo polinomial ao contrário para mostrar que um problema é NP-completo. Vamos avançar com essa ideia e mostrar como poderíamos usar reduções de tempo polinomial para demonstrar que não pode existir nenhum algoritmo de tempo polinomial para um determinado problema B. Suponha que tenhamos um problema de decisão A para o qual já sabemos que não pode existir nenhum algoritmo de tempo polinomial. (Não vamos nos preocupar por enquanto com a maneira de encontrar tal problema A.) Suponha ainda que tenhamos uma redução de tempo polinomial que transforma instâncias de A em instâncias de B. Agora podemos usar uma prova simples por contradição para mostrar que não pode existir nenhum algoritmo de tempo polinomial para B. Suponha o contrário: isto é, suponha que B tenha um algoritmo de tempo polinomial. Então, usando o método mostrado na Figura 34.1 teríamos um modo de resolver o problema A em tempo polinomial, o que contradiz nossa hipótese da inexistência de algoritmo de tempo polinomial para A.

Para a NP-completude não podemos supor que não exista absolutamente nenhum algoritmo de tempo polinomial para o problema A. Contudo, a metodologia da prova é semelhante no sentido de que provamos que o problema B é NP-completo considerando que o problema A também é NP-completo.

Um primeiro problema NP-completo

Como a técnica de redução se baseia em ter um problema que já sabemos ser NP-completo para provar que um problema diferente é NP-completo, precisamos de um "primeiro" problema NP-completo. O problema que usaremos é o da satisfazibilidade de circuitos, no qual temos um circuito combinacional booleano composto por portas AND, OR e NOT, e desejamos saber se existe algum conjunto de entradas booleanas para esse circuito que faça sua saída ser 1. Provaremos que esse primeiro problema é NP-completo na Seção 34.3.

Resumo do capítulo

Este capítulo estuda os aspectos da NP-completude que estão mais diretamente relacionados com a análise de algoritmos. Na Seção 34.1, formalizamos nossa noção de "problema" e definimos a classe de complexidade P de problemas de decisão que podem ser resolvidos em tempo polinomial. Também veremos como essas noções se encaixam na estrutura da teoria de linguagens formais. A Seção 34.2 define a classe NP de problemas de decisão cujas soluções podem ser verificadas em tempo polinomial. Também propõe formalmente a questão P $\neq$ NP.

A Seção 34.3 mostra que podemos relacionar problemas por meio de "reduções" de tempo polinomial, define NP-completude e esboça uma prova de que um problema, denominado "satisfazibilidade de circuitos", é NP-completo. Depois de encontrado um problema NP-completo, mostramos na Seção 34.4 como provar que outros problemas são NP-completos de um modo muito mais simples pela metodologia de reduções. Ilustramos essa metodologia mostrando que dois problemas de satisfazibilidade de fórmulas são NP-completos. Com reduções adicionais, mostramos na Seção 34.5 que vários outros problemas são NP-completos.

34.1 TEMPO POLINOMIAL

Começamos nosso estudo de NP-completude formalizando nossa noção de problemas resolvíveis em tempo polinomial. Em geral, consideramos que esses problemas são tratáveis, mas por razões filosóficas, e não matemáticas. Podemos oferecer três argumentos de sustentação.

O primeiro é que, embora seja razoável considerar como intratável um problema que exige o tempo Q(n_{100}), um número bem pequeno de problemas práticos exige tempo da ordem de um polinômio de grau tão alto. Os problemas calculáveis em tempo polinomial encontrados na prática normalmente exigem um tempo muito menor. A experiência mostrou que, tão logo seja descoberto o primeiro algoritmo de tempo polinomial para um problema, em geral algoritmos mais eficientes vêm logo atrás. Ainda que o melhor algoritmo atual para um problema tenha um tempo de execução de Q(n_{100}), é provável que um algoritmo com um tempo de execução muito melhor logo seja descoberto.

O segundo é que, para muitos modelos razoáveis de computação, um problema que pode ser resolvido em tempo polinomial em um modelo pode ser resolvido em tempo polinomial em outro modelo. Por exemplo, a classe de problemas resolvíveis em tempo polinomial pela máquina de acesso aleatório serial usada na maior parte deste livro é igual à classe de problemas resolvíveis em tempo polinomial em máquinas abstratas de Turing.[1] Também é igual à classe de problemas resolvíveis em tempo polinomial em um computador paralelo quando o número de processadores cresce polinomialmente com o tamanho da entrada.

O terceiro é que a classe de problemas resolvíveis em tempo polinomial tem propriedades de fechamento interessantes, já que os polinômios são fechados por adição, multiplicação e composição. Por exemplo, se a saída de um algoritmo de tempo polinomial é alimentada na entrada de outro, o algoritmo composto é polinomial. O Exercício 34.1-5 pede para mostrar que, se um algoritmo faz um número constante de chamadas a sub-rotinas de tempo polinomial e realiza uma quantidade adicional de trabalho que também leva tempo polinomial, então o tempo de execução do algoritmo composto é polinomial.

Problemas abstratos

Para entender a classe de problemas de tempo polinomial resolvíveis, primeiro devemos ter uma noção formal do que seja um "problema". Definimos um ***problema abstrato*** Q como uma relação binária entre um conjunto I de ***instâncias*** de problemas e um conjunto S de ***soluções*** de problemas. Por exemplo, uma instância de SHORTEST-PATH é uma tripla que consiste em um grafo e dois vértices. Uma solução é uma sequência de vértices no grafo, talvez com a sequência vazia denotando que não existe nenhum caminho. O problema SHORTEST-PATH em si é a relação que associa cada instância de um grafo e dois vértices a um caminho mínimo no grafo que liga os dois vértices. Visto que caminhos mínimos não são necessariamente únicos, uma dada instância de problema pode ter mais de uma solução.

Essa formulação de um problema abstrato é mais geral que o necessário para nossos propósitos. Como vimos antes, a teoria de NP-completude restringe a atenção a ***problemas de decisão***: aqueles que têm uma solução sim/não. Nesse caso, podemos ver um problema de decisão abstrato como uma função que mapeia o conjunto de instâncias I para o conjunto de soluções $\{0, 1\}$. Por exemplo, um problema de decisão relacionado com SHORTEST-PATH é o problema PATH que vimos antes. Se $i = \langle G, u, v, k \rangle$ é uma instância do problema de decisão PATH, então PATH(i) = 1 (sim) se um caminho mínimo de u até v tem no máximo k arestas, e PATH(i) = 0 (não) em caso contrário. Muitos problemas abstratos não são problemas de decisão, mas sim ***problemas de otimização***, que exigem que algum valor seja minimizado ou maximizado. Porém, como vimos anteriormente, em geral podemos reformular um problema de otimização como um problema de decisão que não é mais difícil que o primeiro.

Codificações

Para um programa de computador resolver um problema abstrato, temos de representar as instâncias do problema de um modo que o programa entenda. Uma ***codificação*** de um conjunto S de objetos abstratos é um mapeamento e

de S para o conjunto de cadeias binárias.[2] Por exemplo, todos conhecemos a codificação dos números naturais = {0, 1, 2, 3, 4, ...} como as cadeias {0, 1, 10, 11, 100, ...}. Usando essa codificação, $e(17) = 10001$. Se você já viu representações de computador para caracteres do teclado, provavelmente já viu o código ASCII, pelo qual a codificação de A é 1000001. Podemos codificar um objeto composto como uma cadeia binária combinando as representações de suas partes constituintes. Polígonos, grafos, funções, pares ordenados, programas — todos podem ser codificados como cadeias binárias.

Assim, um algoritmo de computador que "resolve" algum problema de decisão abstrato na realidade toma uma codificação de uma instância de problema como entrada. Denominamos por ***problema concreto*** um problema cujo conjunto de instâncias é o conjunto de cadeias binárias. Dizemos que um algoritmo ***resolve*** um problema concreto no tempo $O(T(n))$ se, dada uma instância i do problema de comprimento $n = |i|$, o algoritmo pode produzir a solução no tempo máximo $O(T(n))$.[3] Portanto, um problema concreto ***é resolvível em tempo polinomial*** se existe um algoritmo para resolvê-lo no tempo $O(n_k)$ para alguma constante k.

Agora podemos definir formalmente a ***classe de complexidade*** **P** como o conjunto de problemas de decisão concretos resolvíveis em tempo polinomial.

Podemos usar codificações para mapear problemas abstratos para problemas concretos. Dado um problema de decisão abstrato Q que mapeia um conjunto de instâncias I para $\{0, 1\}$, uma codificação $e : I \rightarrow \{0, 1\}^*$ pode induzir um problema de decisão concreto relacionado, que denotamos por $e(Q)$.[4] Se a solução para uma instância de problema abstrato $i \in I$ é $Q(i) \in \{0, 1\}$, então a solução para a instância de problema concreto $e(i) \in \{0, 1\}^*$ também é $Q(i)$. Como detalhe técnico, algumas cadeias binárias poderiam representar uma instância de problema abstrato que não é significativa. Por conveniência, convencionamos que qualquer cadeia desse tipo mapeia arbitrariamente para 0. Assim, o problema concreto produz as mesmas soluções que o problema abstrato para instâncias de cadeias binárias que representam as codificações de instâncias do problema abstrato.

Gostaríamos de estender a definição de resolvibilidade em tempo polinomial de problemas concretos para problemas abstratos usando codificações como ponte, mas gostaríamos também que a definição fosse independente de qualquer codificação específica. Isto é, a eficiência da solução de um problema não deve depender do modo como o problema é codificado. Infelizmente, ela depende bastante da codificação. Por exemplo, suponha que um inteiro k deva ser dado como a única entrada para um algoritmo e suponha que o tempo de execução do algoritmo seja $Q(k)$. Se o inteiro k é dado em ***unário*** uma cadeia de k "1s" —, então o tempo de execução do algoritmo é $O(n)$ para entradas de comprimento n, que é tempo polinomial. Todavia, se usarmos a representação binária mais natural do inteiro k, o comprimento da entrada é = n lg $k + 1$. Nesse caso, o tempo de execução do algoritmo é $Q(k) = Q(2_n)$, que é exponencial em relação ao tamanho da entrada. Assim, dependendo da codificação, o algoritmo é executado em tempo polinomial ou em tempo superpolinomial.

A codificação de um problema abstrato é muito importante para nossa compreensão do tempo polinomial. Na realidade, não podemos nem falar em resolver um problema abstrato sem primeiro especificar uma codificação. Apesar disso, na prática, se eliminarmos codificações "dispendiosas" como as unárias, a codificação propriamente dita de um problema fará pouca diferença para o problema ser resolvido ou não em tempo polinomial. Por exemplo, representar inteiros em base 3 em vez de em base 2 (binária) não tem nenhum efeito sobre a solução desse problema em tempo polinomial ou não, já que um inteiro representado em base 3 pode ser convertido em um inteiro representado em base 2 em tempo polinomial.

Dizemos que uma função $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ é ***calculável em tempo polinomial*** se existe um algoritmo de tempo polinomial A que, dada qualquer entrada $x \in \{0, 1\}^*$, produz como saída $f(x)$. Para algum conjunto I de instâncias de problemas, dizemos que duas codificações e_1 e e_2 são ***polinomialmente relacionadas*** se existem duas funções calculáveis em tempo polinomial f_{12} e f_{21} tais que, para qualquer $i \in l$, temos $f_{12}(e_1(i)) = e_2(i)$ e $f_{21}(e_2(i)) = e_1(i)$.[5] Isto é, um algoritmo de tempo polinomial pode calcular a codificação $e_2(i)$ pela codificação $e_1(i)$ e vice-versa. Se duas codificações e_1 e e_2 de um problema abstrato são polinomialmente relacionadas, resolver esse problema em tempo polinomial ou não independe da codificação que usamos, como mostra o lema a seguir.

Lema 34.1

Seja Q um problema de decisão abstrato para um conjunto de instâncias I e sejam e_1 e e_2 codificações polinomialmente relacionadas em I. Então, $e_1(Q) \in$ P se e somente se $e_2(Q) \in$ P.

Prova Basta provar o enunciado na forma em que está, já que a forma inversa é simétrica. Portanto, suponha que e_1 (Q) possa ser resolvida no tempo $O(n_k)$ para alguma constante k. Além disso, suponha que, para qualquer instância de problema i, a codificação $e_1(i)$ possa ser calculada pela codificação $e_2(i)$ no tempo $O(n_c)$ para alguma constante c, onde $n = |e_2(i)|$. Para resolver o problema $e_2(Q)$, para a entrada $e_2(i)$, primeiro calculamos $e_1(i)$ e depois executamos o algoritmo para e_1 (Q) em $e_1(i)$. Quanto tempo isso demora? Converter codificações demora o tempo $O(n_c)$ e, portanto, $|e_2(i)| = O(n_c)$, já que a saída de um computador serial não pode ser mais longa que seu tempo de execução. Resolver o problema para $e_1(i)$ demora o tempo $O(|e_2(i)|^k) = O(n_{ck})$, que é polinomial, já que tanto c quanto k são constantes.

Assim, codificar as instâncias de um problema em binário ou em base 3 não afeta sua "complexidade", isto é, se ele pode ser resolvido em tempo polinomial ou não; porém, se as instâncias forem codificadas em unário, sua complexidade pode mudar. Para que possamos conversar independentemente da codificação, vamos supor em geral que instâncias de problemas estão codificadas em qualquer forma razoável e concisa, a menos que digamos especificamente que não. Para sermos exatos, vamos supor que a codificação de um inteiro está polinomialmente relacionada com sua representação binária e que a codificação de um conjunto finito está polinomialmente relacionada com a sua codificação por meio de uma lista de seus elementos, entre chaves e separados por vírgulas. (ASCII é um desses esquemas de codificação.) Com tal codificação "padrão" em mãos, podemos derivar codificações razoáveis de outros objetos matemáticos, como tuplas, grafos e fórmulas. Para denotar a codificação-padrão de um objeto, colocaremos o objeto entre colchetes angulares. Assim, $\langle G \rangle$ denota a codificação padrão de um grafo G.

Desde que utilizemos implicitamente uma codificação que está polinomialmente relacionada com essa codificação-padrão, podemos conversar diretamente sobre problemas abstratos sem fazer referência a qualquer codificação particular, sabendo que a resolução de um problema abstrato em tempo polinomial não é afetada pela escolha da codificação. Daqui por diante, vamos supor, em geral, que todas as instâncias de problemas são cadeias binárias codificadas segundo a codificação-padrão, a menos que digamos explicitamente que não. Além disso, normalmente negligenciaremos a distinção entre problemas abstratos e concretos. Contudo, o leitor deve ficar atento aos problemas que surgem na prática, nos quais uma codificação-padrão não é óbvia e a codificação faz diferença.

Um arcabouço de linguagens formais

Focalizar problemas de decisão nos permite tirar proveito do mecanismo da teoria das linguagens formais, portanto faremos aqui uma revisão de algumas definições dessa teoria. Um ***alfabeto*** S é um conjunto finito de símbolos. Uma ***linguagem*** L sobre S é qualquer conjunto de cadeias formadas por símbolos extraídos de S. Por exemplo, se S = {0, 1}, o conjunto L = {10, 11, 101, 111, 1011, 1101, 10001, ...} é a linguagem de representações binárias de números primos. Denotamos a ***cadeia vazia*** por , a ***linguagem vazia*** por /0 e a linguagem de todas as cadeias sobre S por S*. Por exemplo, se S = {0, 1}, então S* = {, 0, 1, 00, 01, 10, 11, 000, ...} é o conjunto de todas as cadeias binárias. Toda linguagem L sobre S é um subconjunto de S*.

Podemos efetuar uma variedade de operações em linguagens. Operações da teoria dos conjuntos, como ***união*** e ***interseção***, decorrem diretamente das definições da teoria dos conjuntos. Definimos o ***complemento*** de L por L = S* – L. A ***concatenação*** L_1L_2 de duas linguagens L_1 e L_2 é a linguagem

$$L = \{x_1x_2 : x_1 \in L_1 \text{ e } x_2 \in L_2\} .$$

O ***fechamento*** ou ***estrela de Kleene*** de uma linguagem L é a linguagem

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots ,$$

onde L_k é a linguagem obtida pela concatenação de L com ela mesma k vezes.

Do ponto de vista da teoria das linguagens, o conjunto de instâncias para qualquer problema de decisão Q é simplesmente o conjunto S*, onde S = {0, 1}. Visto que Q é completamente caracterizado pelas instâncias de problema que produzem uma resposta 1 (sim), podemos ver Q como uma linguagem L em S = {0, 1}, onde

$$L = \{x \in \Sigma^* : Q(x) = 1\} .$$

Por exemplo, o problema de decisão PATH tem a linguagem correspondente

$$\begin{aligned} \text{PATH} = \{\langle G, u, v, k\rangle : \; & G = (V, E) \text{ é um grafo não dirigido,} \\ & u, v \in V , \\ & k \geq 0 \text{ é um inteiro, e} \\ & \text{existe um caminho de } u \text{ a } v \text{ em } G \\ & \text{que consiste em no máximo } k \text{ arestas}\} . \end{aligned}$$

(Onde for conveniente, às vezes, usaremos o mesmo nome — PATH nesse caso — para nos referir a um problema de decisão e à sua linguagem correspondente.)

O arcabouço das linguagens formais nos permite expressar concisamente a relação entre problemas de decisão e algoritmos que os resolvem. Dizemos que um algoritmo A ***aceita*** uma cadeia $x \in \{0, 1\}^*$ se, dada a entrada x, a saída do algoritmo $A(x)$ é 1. A linguagem ***aceita*** por um algoritmo A é o conjunto de cadeias $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, isto é, o conjunto de cadeias que o algoritmo aceita. Um algoritmo A ***rejeita*** uma cadeia x se $A(x) = 0$.

Ainda que a linguagem L seja aceita por um algoritmo A, o algoritmo não rejeitará necessariamente uma cadeia $x \notin L$ dada como entrada para ele. Por exemplo, o algoritmo pode entrar em laço para sempre. Uma linguagem L é ***decidida*** por um algoritmo A se toda cadeia binária em L é aceita por A e toda cadeia binária não pertencente a L é rejeitada por A. Uma linguagem L é ***aceita em tempo polinomial*** por um algoritmo A se ela é aceita por A e se, além disso, existe uma constante k tal que, para qualquer cadeia $x \in L$ de comprimento n, o algoritmo A aceita x no tempo $O(n_k)$. Uma linguagem L é ***decidida em tempo polinomial*** por um algoritmo A se existe uma constante k tal que, para qualquer cadeia $x \in \{0, 1\}^*$ de comprimento n, o algoritmo decide corretamente se $x \in L$ no tempo $O(n_k)$. Assim, para aceitar uma linguagem, basta que um algoritmo produza uma resposta quando lhe é dada uma cadeia em L, mas para decidir uma linguagem ele deve aceitar ou rejeitar corretamente toda cadeia em $\{0, 1\}^*$.

Como exemplo, a linguagem PATH pode ser aceita em tempo polinomial. Um algoritmo de aceitação de tempo polinomial verifica se G codifica um grafo não dirigido, verifica se u e v são vértices em G, usa busca em largura para calcular um caminho mínimo de u a v em G e depois compara o número de arestas no caminho mínimo obtido com k. Se G codifica um grafo não dirigido e o caminho de u a v tem no máximo k arestas, o algoritmo produz 1 e para. Caso contrário, o algoritmo roda para sempre. Todavia, esse algoritmo não decide PATH, já que não produz explicitamente 0 para instâncias nas quais um caminho mínimo tem mais de k arestas. Um algoritmo de decisão para PATH deve rejeitar explicitamente cadeias binárias que não pertencem a PATH. Para um problema de decisão como PATH, tal algoritmo de decisão é fácil de projetar: em vez de ser executado para sempre quando não há um caminho de u a v com no máximo k arestas, ele produz 0 e para. No caso de outros problemas, como o problema da parada de Turing, existe um algoritmo de aceitação, mas nenhum algoritmo de decisão.

Podemos definir informalmente uma ***classe de complexidade*** como um conjunto de linguagens cuja pertinência é determinada por uma ***medida de complexidade***, como o tempo de execução, de um algoritmo que determina se dada cadeia x pertence à linguagem L. A definição real de uma classe de complexidade é um pouco mais técnica.[6]

Usando esse arcabouço da teoria das linguagens, podemos dar uma definição alternativa da classe de complexidade P:

$$P = \{L \subseteq \{0,1\}^* : \text{ existe um algoritmo } A \text{ que decide } L \text{ em tempo polinomial}\} .$$

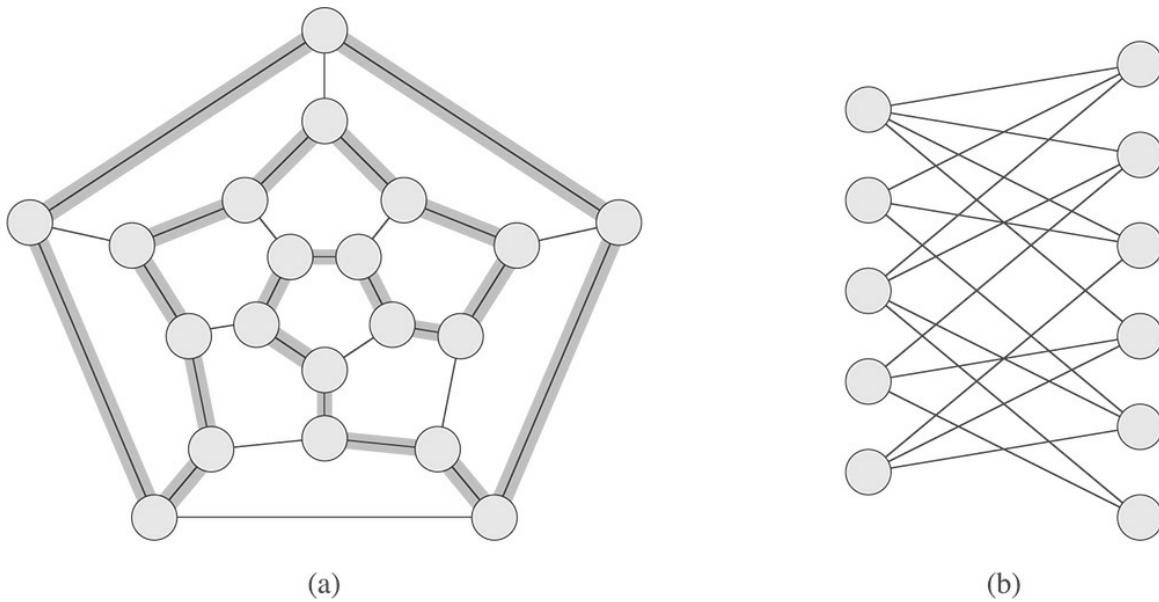
De fato, P também é a classe de linguagens que podem ser aceitas em tempo polinomial.

Teorema 34.2

P = $\{L : L$ é aceita por um algoritmo de tempo polinomial$\}$.

Prova Como a classe de linguagens decidida por algoritmos de tempo polinomial é um subconjunto da classe de linguagens aceita por algoritmos de tempo polinomial, basta mostrar que, se L é aceita por um algoritmo de tempo polinomial, ela é decidida por um algoritmo de tempo polinomial. Seja L a linguagem aceita por algum algoritmo de tempo polinomial A. Usaremos um argumento de "simulação" clássico para construir um outro algoritmo de tempo polinomial A' que decida L. Como A aceita L no tempo $O(n_k)$ para alguma constante k, também existe uma constante c tal que A aceita L em no máximo cn_k etapas. Para qualquer cadeia de entrada x, o algoritmo A' simula cn_k etapas de A. Após simular cn_k etapas, o algoritmo A' inspeciona o comportamento de A. Se A aceitou x, então A' aceita x, produzindo um 1. Se A não aceitou x, então A' rejeita x, produzindo um 0. A sobrecarga da simulação de A por A' não aumenta o tempo de execução de mais de um fator polinomial e, assim, A' é um algoritmo de tempo polinomial que decide L.

Observe que a prova do Teorema 34.2 é não construtiva. Para dada linguagem $L \in$ P, na verdade podemos não conhecer um limite para o tempo de execução para o algoritmo A que aceita L. Apesar disso, sabemos que tal limite existe e, portanto, que existe um algoritmo A' que pode verificar o limite, embora talvez não seja fácil encontrar o algoritmo A'.

Exercícios

34.1-1 Defina o problema de otimização LONGEST-PATH-LENGTH como a relação que associa cada instância de um grafo não dirigido e dois vértices ao número de arestas em um caminho simples de comprimento máximo entre os dois vértices. Defina o problema de decisão LONGEST-PATH = $\{\langle G, u, v, k\rangle : G = (V, E)$ é um grafo não dirigido, $u, v \in V$, $k \geq 0$ é um inteiro e existe um caminho simples de u a v em G que consiste em pelo menos k arestas$\}$. Mostre que o problema de otimização LONGEST-PATH-LENGTH pode ser resolvido em tempo polinomial se e somente se LONGEST-PATH $\in$ P.

34.1-2 Dê uma definição formal para o problema de determinar o ciclo simples de comprimento máximo em um grafo não dirigido. Dê um problema de decisão relacionado. Dê a linguagem correspondente ao problema de decisão.

34.1-3 Dê uma codificação formal de grafos dirigidos como cadeias binárias usando uma representação por matriz de adjacências. Faça o mesmo usando uma representação por lista de adjacências. Demonstre que as duas representações são polinomialmente relacionadas.

34.1-4 O algoritmo de programação dinâmica para o problema da mochila 0-1 que é apresentado no Exercício 16.2-2 é um algoritmo de tempo polinomial? Explique sua resposta.

34.1-5 Mostre que, se um algoritmo faz no máximo um número constante de chamadas a sub-rotinas de tempo polinomial e realiza uma quantidade adicional de trabalho que também demora tempo polinomial, ele é executado em tempo polinomial. Mostre também que um número polinomial de chamadas a sub-rotinas de tempo polinomial pode resultar em um algoritmo de tempo exponencial.

34.1-6 Mostre que a classe P, vista como um conjunto de linguagens, é fechada sob união, interseção, concatenação, complemento e asterisco de Kleene. Isto é, se $L_1, L_2 \in P$, então $L_1 \cup L_2 \in$ P, $L_1 \cap L_2 \in$ P, $L_1 L_2 \in$ P, $L_1 \in$ P e $L_1 \in$ P.

34.2 VERIFICAÇÃO EM TEMPO POLINOMIAL

Agora, examinaremos algoritmos que 'verificam' pertinência a linguagens. Por exemplo, suponha que para uma dada instância $\langle G, u, v, k\rangle$ do problema de decisão PATH, também temos um caminho p de u a v. É fácil verificar se p é um caminho em G e se o comprimento de p é no máximo k e, se for, podemos visualizar p como um "certificado" de que a instância de fato pertence a PATH. Para o problema de decisão PATH, esse certificado não parece nos dar muito. Afinal, PATH pertence a P — de fato, podemos resolver PATH em tempo linear — e, portanto, verificar a pertinência de um determinado certificado demora tanto tempo quanto resolver o problema partindo do zero. Examinaremos agora, um problema para o qual ainda não conhecemos nenhum algoritmo de decisão de tempo polinomia; porém, dado um certificado, a verificação é fácil.

Figura 34.2 (a) Um grafo que representa os vértices, arestas e faces de um dodecaedro, com um ciclo hamiltoniano mostrado por arestas sombreadas. **(b)** Um grafo bipartido com um número ímpar de vértices. Qualquer grafo desse tipo é não hamiltoniano.

Ciclos hamiltonianos

O problema de determinar um ciclo hamiltoniano em um grafo não dirigido é estudado há mais de cem anos. Formalmente, um ***ciclo hamiltoniano*** de um grafo não dirigido $G = (V, E)$ é um ciclo simples que contém cada vértice em V. Um grafo que contém um ciclo hamiltoniano é denominado ***hamiltoniano***; caso contrário, ele é ***não hamiltoniano***. O nome é uma homenagem a W. R. Hamilton, que descreveu um jogo matemático no dodecaedro (Figura 34.2(a)) no qual um jogador fixa cinco alfinetes em quaisquer cinco vértices consecutivos e o outro jogador deve completar o caminho para formar um ciclo que contenha o todos os vértices.[7] O dodecaedro é hamiltoniano, e a Figura 34.2(a) mostra um ciclo hamiltoniano. Contudo, nem todos os grafos são hamiltonianos. Por exemplo, a Figura 34.2(b) mostra um grafo bipartido com um número ímpar de vértices. O Exercício 34.2-2 pede que você mostre que todos esses grafos são não hamiltonianos.

Podemos definir o ***problema do ciclo hamiltoniano***, "Um grafo G tem um ciclo hamiltoniano?", como uma linguagem formal:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ é um grafo hamiltoniano}\}.$$

Como poderia um algoritmo decidir a linguagem HAM-CYCLE? Dada uma instância de problema $\langle G \rangle$, um algoritmo de decisão possível organiza uma lista de todas as permutações dos vértices de G e depois verifica cada permutação para ver se ela é um caminho hamiltoniano. Qual é o tempo de execução desse algoritmo? Se usarmos a codificação "razoável" de um grafo como sua matriz de adjacências, o número m de vértices no grafo será $(\sqrt{n})$, onde $n = |\langle G \rangle|$ é o comprimento da codificação de G. Existem $m!$ permutações possíveis dos vértices, e portanto o tempo de execução é $(m!) = (\sqrt{n}!) = (2\ \sqrt{n}\ )$, que não é $O(n_k)$ para nenhuma constante k. Assim, esse algoritmo ingênuo não é executado em tempo polinomial. Na verdade, o problema do ciclo hamiltoniano é NP-completo, como provaremos na Seção 34.5.

Algoritmos de verificação

Considere um problema ligeiramente mais fácil. Suponha que um amigo lhe diga que um dado grafo G é hamiltoniano e depois se ofereça para provar isso dando a você os vértices em ordem ao longo do ciclo hamiltoniano. Certamente seria bem fácil verificar a prova: basta confirmar que o ciclo dado é hamiltoniano conferindo se ele é uma permutação dos vértices de V e se cada uma das arestas consecutivas ao longo do ciclo existe realmente no grafo. Você certamente poderia implementar esse algoritmo de verificação para ser executado no tempo $O(n_2)$, onde n é o comprimento da codificação de G. Assim, uma prova de que um ciclo hamiltoniano existe em um grafo pode ser verificada em tempo polinomial.

Definimos ***algoritmo de verificação*** como um algoritmo de dois argumentos A, onde um argumento é uma cadeia de entrada comum x e o outro é uma cadeia binária y denominada ***certificado***. Um algoritmo A de dois argumentos ***verifica*** uma cadeia de entrada x se existe um certificado y tal que $A(x, y) = 1$. A ***linguagem verificada*** por um algoritmo de verificação A é

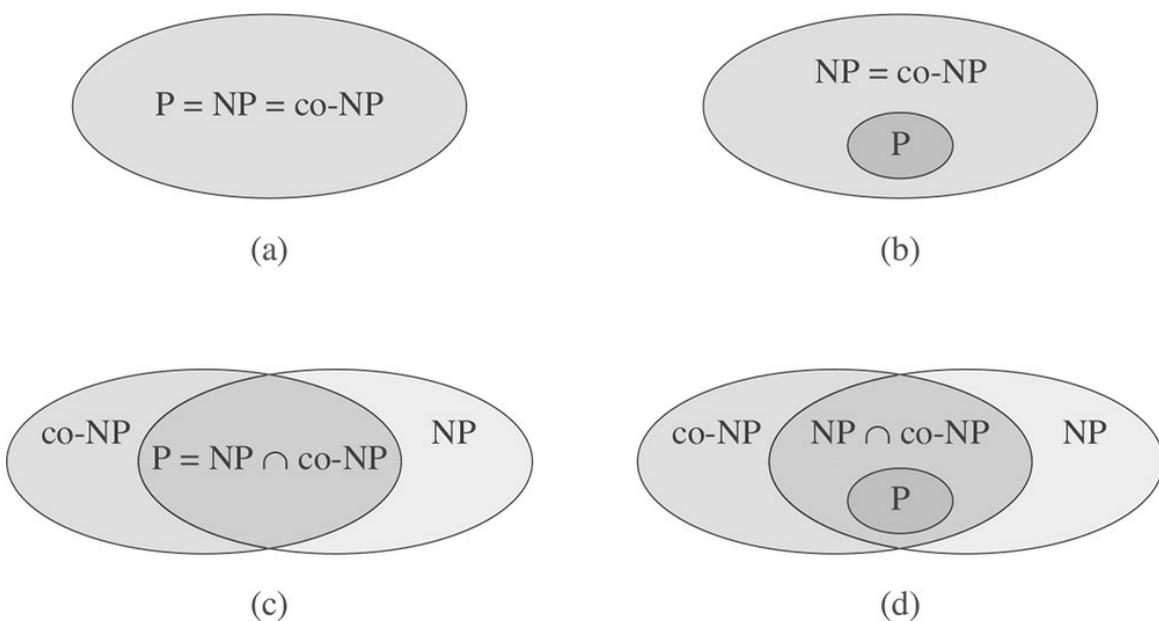
$$L = \{x \in \{0,1\}^* : \text{existe } y \in \{0,1\}^* \text{ tal que } A(x,y) = 1\} .$$

Intuitivamente, um algoritmo A verifica uma linguagem L se, para qualquer cadeia $x \in L$, existe um certificado y que A pode utilizar para provar que $x \in L$. Além disso, para qualquer cadeia $x \notin L$, não deve existir nenhum certificado que prove que $x \in L$. Por exemplo, no problema do ciclo hamiltoniano, o certificado é a lista de vértices em algum ciclo hamiltoniano. Se um grafo é hamiltoniano, o próprio ciclo hamiltoniano oferece informações suficientes para verificar esse fato. Ao contrário, se um grafo não é hamiltoniano, não pode existir nenhuma lista de vértices que engane o algoritmo de verificação fazendo-o acreditar que o grafo é hamiltoniano, já que o algoritmo de verificação examina cuidadosamente o "ciclo" proposto para ter certeza.

A classe de complexidade NP

A ***classe de complexidade NP*** é a classe de linguagens que podem ser verificadas por um algoritmo de tempo polinomial.[8] Mais precisamente, uma linguagem L pertence a NP se e somente se existe um algoritmo de tempo polinomial de duas entradas A e uma constante c tal que

$$L = \{x \in \{0,1\}^* : \text{existe um certificado } y \text{ com } |y| = O(|x|^c) \\ \text{tal que } A(x,y) = 1\} .$$

Dizemos que o algoritmo A ***verifica*** a linguagem L ***em tempo polinomial***.

Pela nossa discussão anterior sobre o problema do ciclo hamiltoniano, agora vemos que HAM-CYCLE $\in$ NP. (É sempre agradável saber que um conjunto importante é não vazio.) Além disso, se $L \in$ P, então $L \in$ NP já que, se

existe um algoritmo de tempo polinomial para decidir L, o algoritmo pode ser facilmente convertido em um algoritmo de verificação de dois argumentos que simplesmente ignora qualquer certificado e aceita exatamente as cadeias de entrada que ele determina que estão em L. Assim, P $\subseteq$ NP.

Não se sabe se P = NP, mas a maioria dos pesquisadores acredita que P e NP não são a mesma classe. Intuitivamente, a classe P consiste em problemas que podem ser resolvidos rapidamente. A classe NP consiste em problemas para os quais uma solução pode ser verificada rapidamente. Você deve ter aprendido por experiência que, muitas vezes, é mais difícil resolver um problema partindo do zero do que verificar uma solução apresentada com clareza, em especial quando se trabalha sob restrições de tempo. Os teóricos da ciência da computação em geral, acreditam que essa analogia se estende às classes P e NP e, por isso, que NP inclui linguagens que não estão em P.

Existe uma evidência mais instigante, porém não conclusiva, de que P $\neq$ NP — a existência de linguagens "NP-completas". Estudaremos essa classe na Seção 34.3.

Figura 34.3 Quatro possibilidades de relações entre classes de complexidade. Em cada diagrama, uma região que envolve uma outra indica uma relação de subconjunto próprio. **(a)** P = NP = co-NP. A maioria dos pesquisadores considera essa possibilidade a mais improvável. **(b)** Se NP é fechada sob complemento, então NP = co-NP, mas não é preciso que seja o caso de P = NP. **(c)** P = NP $\cap$ co-NP, mas NP não é fechada sob complemento. **(d)** NP $\neq$ co-NP e P $\neq$ NP $\cap$ co-NP. A maioria dos pesquisadores considera essa possibilidade a mais provável.

Muitas outras questões fundamentais além da questão P $\neq$ NP permanecem não resolvidas. A Figura 34.3 mostra alguns cenários possíveis. Apesar do grande trabalho de muitos pesquisadores, ninguém sabe sequer se a classe NP é fechada sob complemento. Isto é, $L \in$ NP implica $L \in$ NP? Podemos definir a ***classe de complexidade*** **co-NP** como o conjunto de linguagens L tal que $L \in$ NP. Podemos redefinir a questão de NP ser ou não fechada sob complemento como NP ser ou não igual a co-NP. Visto que P é fechada sob complemento (Exercício 34.1-6), decorre do Exercício 34.2-9 que P $\subseteq$ NP $\cap$ co-NP. Entretanto, mais uma vez, ninguém sabe se P = NP $\cap$ co-NP ou se existe alguma linguagem em NP $\cap$ co-NP – P.

Assim, nossa compreensão da exata relação entre P e NP é terrivelmente incompleta. Apesar disso, e ainda que não consigamos provar que um determinado problema é intratável, se pudermos provar que ele é NP-completo, então já obtivemos valiosa informação sobre ele.

Exercícios

34.2-1 Considere a linguagem GRAPH-ISOMORPHISM = $\{\langle G_1, G_2\rangle : G_1$ e G_2 são grafos isomorfos$\}$. Prove que GRAPH-ISOMORPHISM $\in$ NP, descrevendo um algoritmo de tempo polinomial para verificar a linguagem.

34.2-2 Prove que, se G é um grafo bipartido não dirigido com um número ímpar de vértices, então G é não hamiltoniano.

34.2-3 Mostre que, se HAM-CYCLE $\in$ P, então o problema de organizar uma lista dos vértices de um ciclo hamiltoniano, em ordem, pode ser resolvido em tempo polinomial.

34.2-4 Prove que a classe NP de linguagens é fechada sob união, interseção, concatenação e estrela de Kleene. Discuta o fechamento de NP sob complemento.

34.2-5 Mostre que qualquer linguagem em NP pode ser decidida por um algoritmo executado no tempo $2^{O(n^k)}$ para alguma constante k.

34.2-6 Um ***caminho hamiltoniano*** em um grafo é um caminho simples que visita todo vértice exatamente uma vez. Mostre que a linguagem HAM-PATH = $\{\langle G, u, v\rangle$: existe um caminho hamiltoniano de u até v no grafo $G\}$ pertence a NP.

34.2-7 Mostre que o problema do caminho hamiltoniano do Exercício 34.2-6 pode ser resolvido em tempo polinomial para grafos acíclicos dirigidos. Dê um algoritmo eficiente para o problema.

34.2-8 Seja f uma fórmula booleana construída pelas variáveis de entrada booleanas $x_1, x_2, ..., x_k$, negações ($\neg$), AND ($\wedge$), OR ($\vee$) e parênteses. A fórmula f é uma ***tautologia*** se tem valor 1 para toda atribuição de 1 e 0 às variáveis de entrada. Defina TAUTOLOGY como a linguagem de fórmulas booleanas que são tautologias. Mostre que TAUTOLO- GY $\in$ co-NP.

34.2-9 Prove que P $\subseteq$ co-NP .

34.2-10 Prove que, se NP $\neq$ co-NP, então P $\neq$ NP.

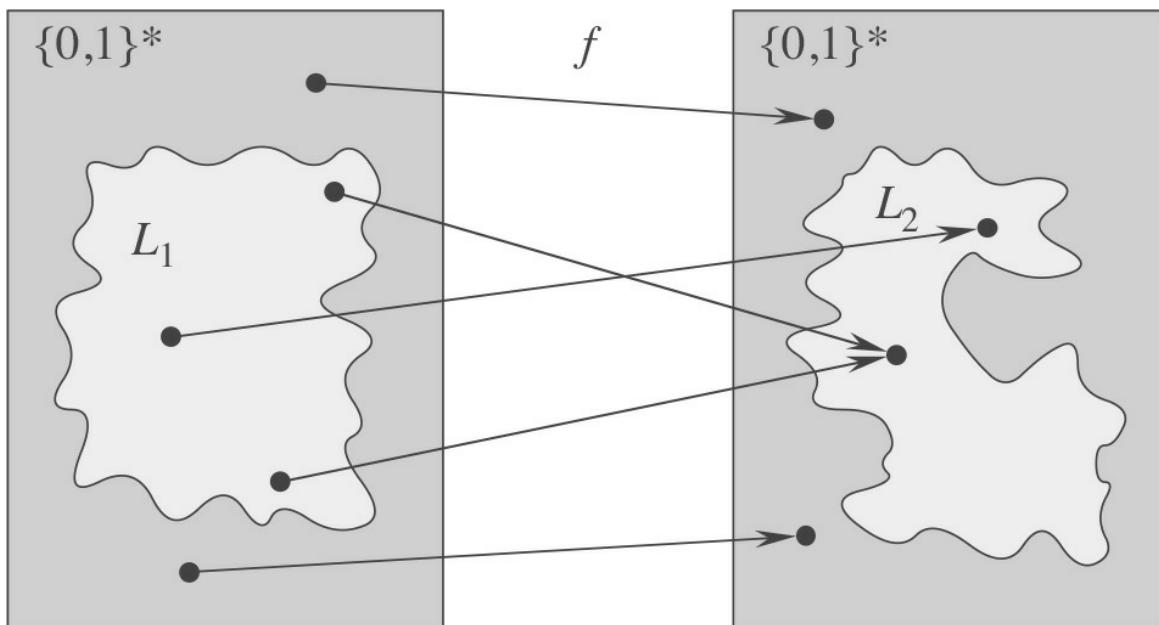
34.2-11 Seja G um grafo conexo não dirigido com pelo menos três vértices e seja G_3 o grafo obtido ligando-se todos os pares de vértices que estão conectados por um caminho em G de comprimento máximo 3. Prove que G_3 é hamiltoniano. (*Sugestão*: Construa uma árvore geradora para G e use um argumento indutivo.)

34.3 NP-COMPLETUDE E REDUTIBILIDADE

Talvez a razão mais forte pela qual os teórico da ciência da computação acreditem que P $\neq$ NP seja a existência da classe de problemas “NP-completos”. Essa classe tem a intrigante propriedade de que, se *algum* problema NP-completo pode ser resolvido em tempo polinomial, então *todo* problema em NP tem uma solução em tempo polinomial, isto é, P = NP. Entretanto, apesar de anos de estudo, nenhum algoritmo de tempo polinomial jamais foi descoberto para qualquer problema NP-completo.

A linguagem HAM-CYCLE é um problema NP-completo. Se pudéssemos decidir HAM-CYCLE em tempo polinomial, poderíamos resolver todo problema em NP em tempo polinomial. De fato, se NP – P viesse a ser não vazia, poderíamos dizer com certeza que HAM-CYCLE $\in$ NP – P.

As linguagens NP-completas são, em certo sentido, as linguagens “mais difíceis” em NP. Nesta seção, mostraremos como comparar a “dificuldade” relativa de linguagens usando uma noção precisa denominada “redutibilidade em tempo polinomial”. Depois, definimos formalmente as linguagens NP-completas e terminamos

esboçando uma prova de que uma dessas linguagens, denominada CIRCUIT-SAT, é NP-completa. Nas Seções 34.4 e 34.5, usaremos a noção de redutibilidade para mostrar que muitos outros problemas são NP-completos.

Redutibilidade

Intuitivamente, um problema Q pode ser reduzido a outro problema Q' se qualquer instância de Q pode ser "facilmente reformulada" como uma instância de Q', cuja solução dá uma solução para a instância de Q. Por exemplo, o problema de resolver equações lineares em um x indeterminado se reduz ao problema de resolver equações quadráticas. Dada uma instância $ax + b = 0$, transformamos essa instância em $0x_2 + ax + b = 0$, cuja solução dá uma solução para $ax + b = 0$. Assim, se um problema Q se reduz a um outro problema Q', então Q não é, em certo sentido, "mais difícil de resolver" que Q'.

Retornando à nossa estrutura de linguagens formais para problemas de decisão, dizemos que uma linguagem L_1 é ***redutível em tempo polinomial*** a uma linguagem L_2, escrita $L_1 \leq_P L_2$, se existe uma função calculável de tempo polinomial $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ tal que, para todo $x \in \{0, 1\}^*$,

$$x \in L_1 \text{ se e somente se } f(x) \in L_2. \tag{34.1}$$

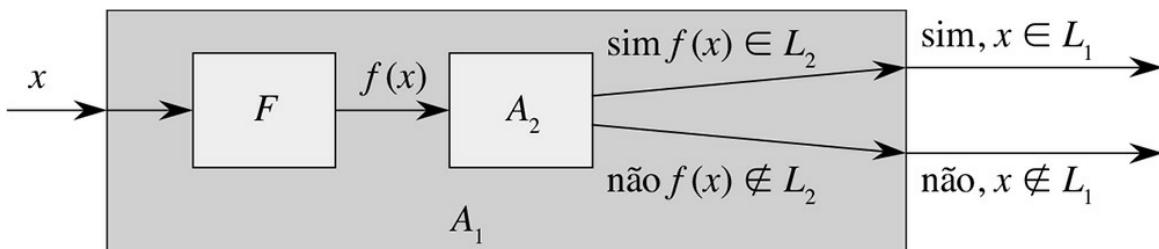
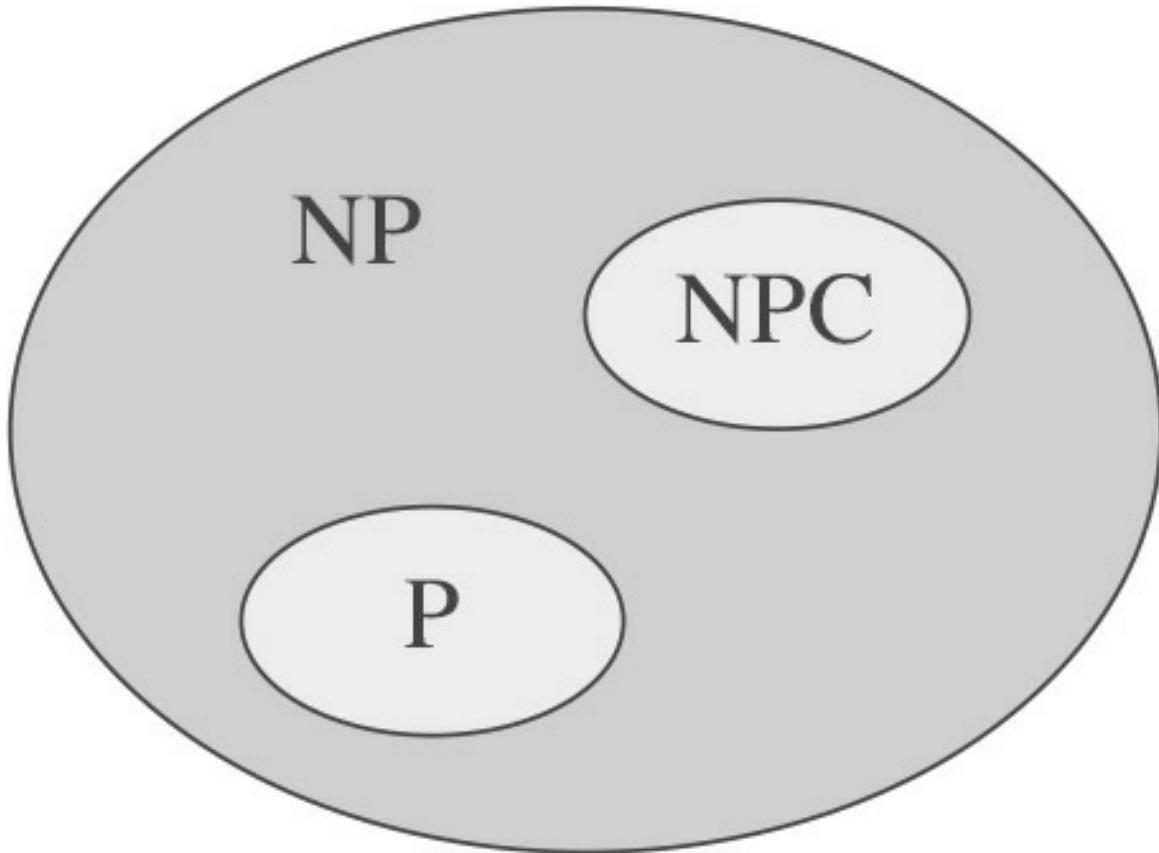
Denominamos a função f ***função redução***, e um algoritmo de tempo polinomial F que calcula f é denominado ***algoritmo de redução***.

Figura 34.4 Ilustração de uma redução de tempo polinomial de uma linguagem L_1 a uma linguagem L_2 por meio de uma função de redução f. Para qualquer entrada $x \in \{0, 1\}^*$, a questão de saber se $x \in L_1$ tem a mesma resposta que a questão de saber se $f(x) \in L_2$.

A Figura 34.4 ilustra a ideia de uma redução de tempo polinomial de uma linguagem L_1 a uma outra linguagem L_2. Cada linguagem é um subconjunto de $\{0, 1\}^*$. A função redução f dá um mapeamento em tempo polinomial tal que, se $x \in L_1$, então $f(x) \in L_2$. Além disso, se $x \notin L_1$, então $f(x) \notin L_2$. Assim, a função redução mapeia qualquer instância x do problema de decisão representado pela linguagem L_1 para uma instância $f(x)$ do problema representado por L_2. Dar uma resposta à questão "$f(x) \in L_2$ ou não" dá diretamente a resposta à questão "$x \in L_1$ ou não".

Reduções de tempo polinomial nos dão uma poderosa ferramenta para provar que diversas linguagens pertencem a P.

Lema 34.3

Se $L_1, L_2 \subseteq \{0, 1\}^*$ são linguagens tais que $L_1 \leq_P L_2$, então $L_2 \in$ P implica $L_1 \in$ P.

Prova Seja A_2 um algoritmo de tempo polinomial que decide L_2 e seja F um algoritmo de redução de tempo polinomial que calcula a função redução f. Construiremos um algoritmo de tempo polinomial A_1 que decide L_1.
A Figura 34.5 ilustra a construção de A_1. Para uma dada entrada $x \in \{0, 1\}^*$, o algoritmo A_1 usa F para transformar x em $f(x)$ e depois usa A_2 para testar se $f(x) \in L_2$. O algoritmo A_1 toma a saída do algoritmo A_2 e produz aquela resposta como a sua própria resposta.
A correção de A_1 decorre da condição (34.1). O algoritmo é executado em tempo polinomial, já que tanto F quanto A_2 são executados em tempo polinomial (ver o Exercício 34.1-5).

NP-completude

Reduções de tempo polinomial proporcionam um meio formal de mostrar que um problema é no mínimo tão difícil quanto um outro, a menos de um fator de tempo polinomial. Isto é, se $L_1 \leq_P L_2$, então L_1 não é mais do que um fator polinomial mais difícil que L_2, e esse é o motivo por que a notação "menor que ou igual a" para redução é mnemônica. Agora podemos definir o conjunto de linguagens NP-completas, que são os problemas mais difíceis em NP.

Figura 34.5 A prova do Lema 34.3. O algoritmo F é um algoritmo de redução que calcula a função redução f de L_1 a L_2 em tempo polinomial, e A_2 é um algoritmo de tempo polinomial que decide L_2. O algoritmo A_1 decide se $x \in L_1$ usando F para transformar qualquer entrada x em $f(x)$ e depois usando A_2 para decidir se $f(x) \in L_2$.

Figura 34.6 Como a maioria dos teóricos da ciência da computação vê as relações entre P, NP e NPC. Tanto P quanto NPC estão inteiramente contidas em NP, e $P \cap NPC = \emptyset$.

Uma linguagem $L \subseteq \{0, 1\}^*$ é ***NP-completa*** se

1. $L \in$ NP e
2. $L' \leq_P L$ para todo $L' \in$ NP.

Se uma linguagem L satisfaz a propriedade 2 mas não necessariamente a propriedade 1, dizemos que L é ***NP-difícil***. Também definimos NPC como a classe de linguagens NP-completas.

Como mostra o teorema a seguir, NP-completude é o ponto crucial para decidir se P é de fato igual a NP.

Teorema 34.4

Se algum problema NP-completo é resolvível em tempo polinomial, então P = NP. Equivalentemente, se algum problema em NP não é resolvível em tempo polinomial, nenhum problema NP-completo é resolvível em tempo polinomial.

Prova Suponha que $L \in$ P e também que $L \in$ NPC. Para qualquer $L' \in$ NP, temos $L' \leq_P L$ pela propriedade 2 da definição de NP-completude. Assim, pelo Lema 34.3, também temos que $L' \in$ P, o que prova o primeiro enunciado do teorema.

Para provar o segundo enunciado, observe que ele é o contrapositivo do primeiro enunciado.

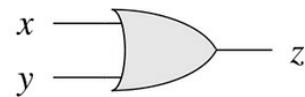
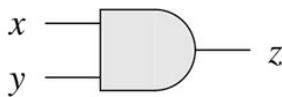
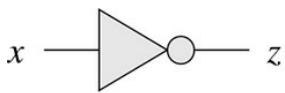
É por essa razão que a pesquisa da questão P ≠ NP se concentra em torno dos problemas NP-completos. A maioria dos teóricos da ciência da computação acredita que P ≠ NP, o que leva às relações entre P, NP e NPC mostradas na Figura 34.6. Porém, que saibamos, bem que alguém ainda pode apresentar um algoritmo de tempo polinomial para um problema NP-completo e assim provar que P = NP. Não obstante, como ainda não foi descoberto

nenhum algoritmo de tempo polinomial para qualquer problema NP-completo, uma prova de que um problema é NP-completo dá uma excelente evidência de que ele é intratável.

Satisfazibilidade de circuitos

Definimos a noção de um problema NP-completo mas, até este ponto, não provamos realmente que nenhum problema é NP-completo. Uma vez provado que pelo menos um problema é NP-completo, poderemos usar redutibilidade de tempo polinomial como uma ferramenta para provar que outros problemas são NP-completos. Assim, agora focalizamos a demonstração da existência de um problema NP-completo: o problema da satisfazibilidade de circuitos.

Infelizmente, a prova formal de que o problema da satisfazibilidade de circuitos é NP-completo requer detalhes técnicos que estão fora do escopo deste texto. Em vez disso, descreveremos informalmente uma prova que depende de um entendimento básico de circuitos combinacionais booleanos.

Circuitos combinacionais booleanos são construídos com elementos combinacionais booleanos interconectados por fios. Um ***elemento combinacional booleano*** é qualquer elemento de circuito que tem um número constante de entradas e saídas booleanas e que executa uma função bem definida. Valores booleanos são extraídos do conjunto {0, 1}, onde 0 representa FALSE e 1 representa TRUE.

Os elementos combinacionais booleanos que utilizamos no problema da satisfazibilidade de circuitos calculam uma função booleana simples e são conhecidos como, ***portas lógicas***. A Figura 34.7 mostra as três portas lógicas básicas que usamos no problema da satisfazibilidade de circuitos: a ***porta NOT*** (ou ***inversora***), a ***porta AND*** e a ***porta OR***. A porta NOT toma uma única ***entrada*** binária x, cujo valor é 0 ou 1, e produz uma ***saída*** binária z cujo valor é o oposto do valor da entrada. Cada uma das outras duas portas toma duas entradas binárias x e y e produz uma única saída binária z.

Podemos descrever o funcionamento de cada porta e de qualquer elemento combinacional booleano por uma ***tabela verdade***, mostrada sob cada porta na Figura 34.7. Uma tabela verdade dá as saídas do elemento combinacional para cada configuração possível das entradas. Por exemplo, a tabela verdade para a porta OR informa que, quando as entradas são $x = 0$ e $y = 1$, o valor da saída é $z = 1$. Usamos os símbolos $\neg$ para denotar a função NOT, $\wedge$ para denotar a função AND e $\vee$ para denotar a função OR. Assim, por exemplo, $0 \vee 1 = 1$.

Podemos generalizar as portas AND e OR para que tomem mais de duas entradas. A saída de uma porta AND é 1 se todas as suas entradas são 1, caso contrário sua saída é 0. A saída de uma porta OR é 1 se qualquer de suas entradas é 1, caso contrário sua saída é 0.

Um ***circuito combinacional booleano*** consiste em um ou mais elementos combinacionais booleanos interligados por ***fios***. Um fio pode ligar a saída de um elemento à entrada de outro, fornecendo assim o valor de saída do primeiro elemento como um valor de entrada do segundo. A Figura 34.8 mostra dois circuitos combinacionais booleanos semelhantes, diferentes somente por uma porta. A parte (a) da figura também mostra os valores nos fios individuais, dada a entrada $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$. Embora um único fio não possa ter mais de uma saída de elemento combinacional ligada a ele, esse fio pode alimentar várias entradas de elementos. O número de entradas alimentadas por um fio é denominado ***leque de saída*** do fio. Se nenhuma saída de elemento está ligada a um certo fio, esse fio é uma ***entrada de circuito***, que aceita valores de entrada de uma fonte externa. Se nenhuma entrada de elemento está ligada a um fio, esse fio é uma ***saída de circuito***, que fornece os resultados da computação do circuito para o mundo exterior. (Um fio interno também pode ter uma saída em leque para uma saída de circuito.) Para definir o problema da satisfazibilidade de circuitos, limitamos o número de saídas de circuitos a 1, embora no projeto de hardware propriamente dito um circuito combinacional booleano possa ter várias saídas.

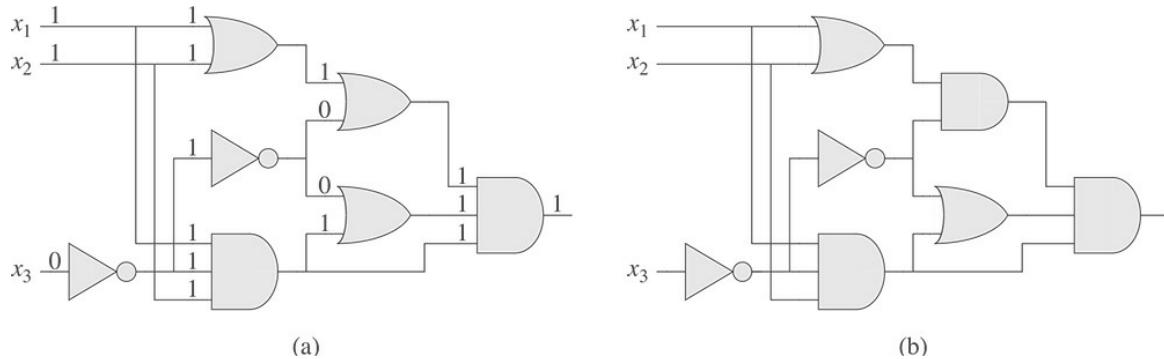
Circuitos combinacionais booleanos não contêm nenhum ciclo. Em outras palavras, suponha que criamos um grafo dirigido $G = (V, E)$ com um vértice para cada elemento combinacional e com k arestas dirigidas para cada fio cuja saída em leque é k; o grafo contém uma aresta dirigida (u, v) se um fio liga a saída do elemento u a uma entrada de elemento v. Então, G deve ser acíclico.

x	$\neg x$
0	1
1	0

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Figura 34.7 Três portas lógicas básicas, com entradas e saídas binárias. Sob cada porta está a tabela verdade que descreve a operação da porta. **(a)** A porta NOT. **(b)** A porta AND. **(c)** A porta OR.

Uma ***atribuição verdade*** para um circuito combinacional booleano é um conjunto de valores de entrada booleanos. Dizemos que um circuito combinacional booleano de uma saída é ***satisfazível*** se tem uma ***atribuição que satisfaz***: uma atribuição verdade que faz com que a saída do circuito seja 1. Por exemplo, o circuito na Figura 34.8(a) tem a atribuição que satisfaz $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ e, assim, é satisfazível. Como o Exercício 34.3-1 pede para mostrar, nenhuma atribuição de valores para x_1, x_2 e x_3 faz com que o circuito da Figura 34.8(b) produza uma saída 1; ele sempre produz 0 e, assim, é insatisfazível.

O ***problema da satisfazibilidade de circuitos*** é: "Dado um circuito combinacional booleano composto de portas AND, OR e NOT, ele é satisfazível?" Contudo, para propor formalmente essa pergunta, temos de concordar com uma codificação-padrão para circuitos. O ***tamanho*** de um circuito combinacional booleano é o número de elementos combinacionais somado ao número de fios no circuito. Poderíamos criar uma codificação semelhante a um grafo que mapeie qualquer circuito C dado para uma cadeia binária $\langle C\rangle$ cujo comprimento é polinomial em relação ao tamanho do próprio circuito. Então, como uma linguagem formal, podemos definir CIRCUIT-SAT = $\{ \langle C\rangle : C$ é um circuito combinacional booleano satisfazível$\}$.

O problema da satisfazibilidade de circuitos surge na área da otimização de hardware auxiliada por computador. Se um subcircuito sempre produz 0, esse subcircuito é desnecessário; o projetista pode substituí-lo por um subcircuito mais simples que omite todas as portas lógicas e produz o valor constante 0 como sua saída. Você pode ver por que gostaríamos de ter um algoritmo de tempo polinomial para esse problema.

Dado um circuito C, poderíamos tentar determinar se ele é satisfazível apenas verificando todas as atribuições possíveis para as entradas. Infelizmente, se o circuito tem k entradas, há 2^k atribuições possíveis. Quando o tamanho de C é polinomial em k, verificar cada uma demora o tempo (2^k), que é superpolinomial em relação ao tamanho do circuito.[9] De fato, como já afirmamos, há fortes evidências de que não existe nenhum algoritmo de tempo polinomial que resolva o problema da satisfazibilidade de circuitos porque a satisfazibilidade de circuitos é um problema NP-completo. Dividimos a prova desse fato em duas partes, com base nas duas partes da definição de NP-completude.

Lema 34.5

O problema da satisfazibilidade de circuitos pertence à classe NP.

Figura 34.8 Duas instâncias do problema da satisfazibilidade de circuitos. **(a)** A atribuição $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ para as entradas desse circuito faz com que a saída do circuito seja 1. O circuito é então satisfazível. **(b)** Nenhuma atribuição para as entradas desse circuito pode fazer com que a saída do circuito seja 1. Então, o circuito é insatisfazível.

Prova Daremos um algoritmo A de tempo polinomial com duas entradas que pode verificar CIRCUIT-SAT. Uma das entradas para A é (uma codificação-padrão de) um circuito combinacional booleano C. A outra entrada é um certificado que corresponde a uma atribuição de valores booleanos aos fios em C (veja no Exercício 34.3-4 um certificado menor).

Construímos o algoritmo A da maneira mostrada a seguir. Para cada porta lógica no circuito, ele verifica se o valor fornecido pelo certificado no fio de saída é calculado corretamente em função dos valores nos fios de entrada. Então, se a saída do circuito inteiro é 1, o algoritmo produz 1, já que os valores atribuídos às entradas de C fornecem uma atribuição que satisfaz. Caso contrário, A produz 0.

Sempre que um circuito satisfazível C é dado como entrada para o algoritmo A, existe um certificado cujo comprimento é polinomial em relação ao tamanho de C e que faz com que A produza um 1. Sempre que um circuito insatisfazível é dado como entrada, nenhum certificado pode enganar A e fazê-lo acreditar que o circuito é satisfazível. O algoritmo A é executado em tempo polinomial: com uma boa implementação, o tempo linear é suficiente. Assim, podemos verificar CIRCUIT-SAT em tempo polinomial, e CIRCUIT-SAT $\in$ NP.

A segunda parte da prova de que CIRCUIT-SAT é NP-completo é mostrar que a linguagem é NP-difícil. Isto é, devemos mostrar que toda linguagem em NP é redutível em tempo polinomial a CIRCUIT-SAT. A prova propriamente dita desse fato é repleta de complexidades técnicas, portanto nos contentaremos com um esboço da prova baseado em alguma compreensão do funcionamento interno do hardware de computadores.

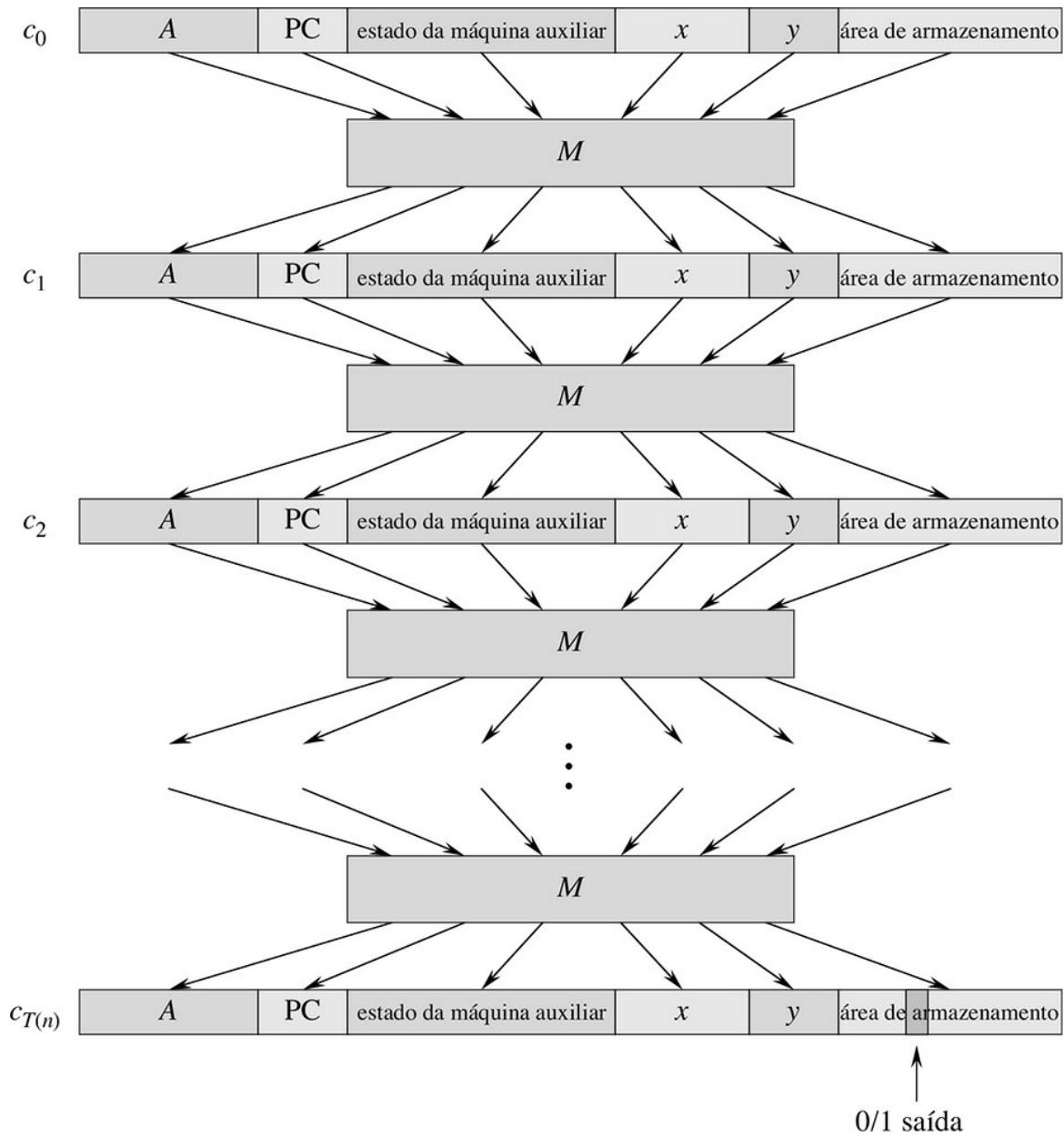
Um programa de computador é armazenado na memória do computador como uma sequência de instruções. Uma instrução típica codifica uma operação a ser executada, endereços de operandos na memória e um endereço onde o resultado deve ser armazenado. Uma posição especial de memória, denominada ***contador de programa***, controla qual instrução deve ser executada em seguida. O contador de programa é incrementado automaticamente sempre que uma instrução é recuperada, o que faz o computador executar instruções sequencialmente. Contudo, a execução de uma instrução pode fazer com que um valor seja escrito no contador de programa, o que altera a execução sequencial normal e permite que o computador execute laços e desvios condicionais.

Em qualquer ponto durante a execução de um programa, a memória do computador guarda todo o estado da computação. (Entendemos que a memória inclui o programa em si, o contador de programa, a área de armazenamento e quaisquer dos vários bits de estado que um computador mantém para contabilidade.) Denominamos por ***configuração*** qualquer estado particular da memória do computador. Podemos ver que a execução de uma instrução pode ser vista como o mapeamento de uma configuração para outra. O hardware do computador que executa esse mapeamento pode ser implementado como um circuito combinacional booleano, que denotamos por M na prova do lema a seguir.

Lema 34.6

O problema da satisfazibilidade de circuitos é NP-difícil.

Prova Seja L qualquer linguagem em NP. Descreveremos um algoritmo de tempo polinomial F que calcula uma função redução f que mapeia toda cadeia binária x para um circuito $C = f(x)$ tal que $x \in L$ se e somente se $C \in$ CIRCUIT-SAT.

Visto que $L \in$ NP, deve existir um algoritmo A que verifica L em tempo polinomial. O algoritmo F que construiremos usará o algoritmo A de duas entradas para calcular a função redução f.

Seja $T(n)$ o tempo de execução do pior caso do algoritmo A para cadeias de entrada de comprimento n e seja $k \geq 1$ uma constante tal que $T(n) = O(n_k)$ e o comprimento do certificado é $O(n_k)$. (O tempo de execução de A é na realidade um polinômio no tamanho total da entrada, o que inclui uma cadeia de entrada, bem como um certificado; porém, como o comprimento do certificado é polinomial no comprimento n da cadeia de entrada, o tempo de execução é polinomial em n.)

A ideia básica da prova é representar a computação de A como uma sequência de configurações. Como mostra a Figura 34.9, podemos dividir cada configuração em partes que consistem no programa para A, contador de programa e estado da máquina auxiliar, entrada x, certificado y e área de armazenamento. O circuito combinacional M, que implementa o hardware do computador, mapeia cada configuração c_i para a próxima configuração $c_i + {}^1$, começando da configuração inicial $c_{o.}$ O algoritmo A grava sua saída — 0 ou 1 — em alguma localização designada quando termina de executar e, se considerarmos que dali em diante A para, o valor nunca muda. Assim, se o algoritmo é executado durante no máximo $T(n)$ etapas, a saída aparece como um dos bits em $c_T(_n)$.

O algoritmo de redução F constrói um único circuito combinacional que calcula todas as configurações produzidas por uma dada configuração inicial. A ideia é colar $T(n)$ cópias do circuito M. A saída do i-ésimo circuito, que produz a configuração c_i, alimenta diretamente a entrada do($i + 1$)-ésimo circuito. Assim, as configurações, em vez de serem armazenadas na memória do computador, simplesmente permanecem como valores nos fios que ligam cópias de M.

Lembre-se do que o algoritmo de redução de tempo polinomial F deve fazer. Dada uma entrada x, ele tem de calcular um circuito $C = f(x)$ que é satisfazível se e somente se existe um certificado y tal que $A(x, y) = 1$. Quando F obtém uma entrada x, primeiro ele calcula $n = |x|$ e constrói um circuito combinacional C' que consiste em $T(n)$ cópias de M. A entrada para C' é uma configuração inicial correspondente a uma computação em $A(x, y)$, e a saída é a configuração $c_T(_n)$.

Figura 34.9 A sequência de configurações produzidas por um algoritmo A que executa uma entrada x e um certificado y. Cada configuração representa o estado do computador para uma etapa da computação e, além de A, x e y, inclui o contador de programa (PC), o estado da máquina auxiliar e a área de armazenamento. Exceto pelo certificado y, a configuração inicial c_0 é constante. Um circuito combinacional booleano M mapeia cada configuração para a configuração seguinte. A saída é um bit distinto na área de armazenamento.

O algoritmo F modifica ligeiramente o circuito C' para construir o circuito $C = f(x)$. Primeiro, ele liga as entradas para C' correspondentes ao programa para A, o contador de programa inicial, a entrada x e o estado inicial da memória diretamente a esses valores conhecidos. Assim, as únicas entradas restantes para o circuito correspondem ao certificado y. Depois, ignora todas as saídas do circuito, exceto o de $c_T(_n)$ que corresponde à saída de A. Esse circuito C assim construído calcula $C(y) = A(x, y)$ para qualquer entrada y de comprimento $O(n_k)$. O algoritmo de redução F, quando recebe uma cadeia de entrada x, calcula esse circuito C e o produz como saída.

Precisamos provar duas propriedades. Em primeiro lugar, devemos mostrar que F calcula corretamente uma função redução f. Isto é, temos de mostrar que C é satisfazível se e somente se existe um certificado y tal que $A(x, y) =$ 1. Em segundo lugar, devemos mostrar que F é executado em tempo polinomial.

Para mostrar que F calcula corretamente uma função redução, vamos supor que exista um certificado y de comprimento $O(n_k)$ tal que $A(x, y) = 1$. Então, se aplicarmos os bits de y às entradas de C, a saída de C é $C(y) = A(x,$

y) = 1. Assim, se existe um certificado, C é satisfazível. Ao contrário, suponha que C seja satisfazível. Consequentemente, existe uma entrada y para C tal que $C(y) = 1$, do que concluímos que $A(x, y) = 1$. Assim, F calcula corretamente uma função redução. Para completar a prova, basta mostrar que F é executado em tempo polinomial em $n = |x|$.

A primeira observação que fazemos é que o número de bits necessários para representar uma configuração é polinomial em n. O programa para A em si tem tamanho constante, independentemente do comprimento de sua entrada x. O comprimento da entrada x é n, e o comprimento do certificado y é $O(n_k)$. Visto que o algoritmo é executado para no máximo $O(n_k)$ etapas, a quantidade de área de armazenamento exigida por A também é polinomial em n. (Supomos que essa memória seja contígua; o Exercício 34.3-5 pede para estender o argumento à situação na qual as posições acessadas por A estão espalhadas por uma região de memória muito maior e que o padrão de espalhamento específico pode ser diferente para cada entrada x.)

O circuito combinacional M que implementa o hardware do computador tem tamanho polinomial no comprimento de uma configuração, que é $O(n_k)$; por consequência, o tamanho de M é polinomial em n. (A maior parte desses circuitos implementa a lógica do sistema de memória.) O circuito C consiste em no máximo $t = O(n_k)$ cópias de M e, consequentemente, tem tamanho polinomial em n. O algoritmo de redução F pode construir C a partir de x em tempo polinomial, já que cada etapa da construção demora tempo polinomial.

Portanto, a linguagem CIRCUIT-SAT é no mínimo tão difícil quanto qualquer linguagem em NP e, visto que pertence a NP, ela é NP-completa.

Teorema 34.7

O problema da satisfazibilidade de circuitos é NP-completo.

Prova Imediata, pelos Lemas 34.5 e 34.6 e pela definição de NP-completude.

Exercícios

34.3-1 Confirme que o circuito da Figura 34.8(b) é insatisfazível.

34.3-2 Mostre que a relação $\leq_P$ é uma relação transitiva em linguagens. Isto é, mostre que, se $L_1 \leq_P L_2$ e $L_2 \leq_P L_3$, então $L_1 \leq_P L_3$.

34.3-3 Prove que $L \leq_P L$ se e somente se $L \leq_P L$.

34.3-4 Mostre que poderíamos ter usado uma atribuição satisfatória como um certificado em uma prova alternativa do Lema 34.5. Qual certificado facilita a prova?

34.3-5 A prova do Lema 34.6 supõe que a área de armazenamento para o algoritmo A ocupa uma região contígua de tamanho polinomial. Em que parte exploramos essa hipótese? Demonstre que essa hipótese não envolve qualquer perda de generalidade.

34.3-6 Uma linguagem L é ***completa*** para uma classe de linguagem C com relação a reduções de tempo polinomial se $L \in C$ e $L' \leq_P L$ para todo $L' \in C$. Mostre que /0 e $\{0, 1\}^*$ são as únicas linguagens em P que não são completas para P com relação a reduções de tempo polinomial.

34.3-7 Mostre que, no que diz respeito a reduções de tempo polinomial (veja o Exercício 34.3- 6), L é completa para NP se e somente se L é completa para co-NP.

34.3-8 O algoritmo de redução F na prova do Lema 34.6 constrói o circuito $C = f(x)$ com base no conhecimento de x, A e k. O professor Sartre observa que a cadeia x é uma entrada para F, mas somente a existência de A, k e do fator constante implícito no tempo de execução $O(n_k)$ é conhecida para F (já que a linguagem L pertence a NP), e não seus valores reais. Assim, o professor conclui que F não pode construir o circuito C e que a linguagem CIRCUIT-SAT não é necessariamente NP-difícil. Explique a falha no raciocínio do professor.

34.4 Provas da NP-completude

Provamos que o problema da satisfazibilidade de circuitos é NP-completo por uma prova direta de que $L \leq_P$ CIRCUIT-SAT para toda linguagem $L \in$ NP. Nesta seção, mostraremos como provar que as linguagens são NP-completas sem reduzir diretamente *toda* linguagem em NP à linguagem dada. Ilustraremos essa metodologia provando que vários problemas de satisfazibilidade de fórmulas são NP-completos. A Seção 34.5 dá muitos outros exemplos da metodologia.

O lema a seguir é a base de nosso método para mostrar que uma linguagem é NP-completa.

Lema 34.8

Se L é uma linguagem tal que $L' \leq_P L$ para alguma $L' \in$ NPC, então L é NP-difícil. Se, além disso, $L \in$ NP, então $L \in$ NPC.

Prova Visto que L' é NP-completa, para todo $L'' \in$ NP, temos $L'' \leq_P L'$. Por hipótese, $L' \leq_P L$ e, assim, por transitividade (Exercício 34.3-2), temos $L'' \leq_P L$, o que mostra que L é NP-difícil. Se $L \in$ NP, também temos $L \in$ NPC.

Em outras palavras, reduzindo a L uma linguagem NP-completa L' conhecida, reduzimos implicitamente toda linguagem em NP a L. Assim, o Lema 34.8 apresenta um método para provar que uma linguagem L é NP-completa:

1. Prove que $L \in$ NP.
2. Selecione uma linguagem NP-completa conhecida L'.
3. Descreva um algoritmo que calcule uma função f mapeando toda instância $x \in \{0, 1\}^*$ de L' para uma instância $f(x)$ de L.
4. Prove que a função f satisfaz a $x \in L'$ se e somente se $f(x) \in L$ para todo $x \in \{0, 1\}^*$.
5. Prove que o algoritmo que calcula f é executado em tempo polinomial.

(As etapas 2 a 5 mostram que L é NP-difícil.) Essa metodologia de redução a partir de uma única linguagem NP-completa conhecida é muitíssimo mais simples que o processo mais complicado de mostrar diretamente como reduzir partindo de toda linguagem em NP. Provar que CIRCUIT-SAT $\in$ NPC já entreabriu a porta. Como sabemos que a satisfazibilidade de circuitos é um problema NP-completo, agora podemos provar com muito mais facilidade que outros problemas são NP-completos. Além disso, à medida que desenvolvermos um catálogo de problemas NP-completos conhecidos, teremos cada vez mais opções de linguagens a partir das quais reduzir.

Satisfazibilidade de fórmulas

Ilustramos a metodologia de redução dando uma prova de NP-completude para o problema de determinar se uma fórmula booleana, não um circuito, é satisfazível. Esse problema tem a honra histórica de ter sido o primeiro problema a ser apresentado como NP-completo.

Formulamos o problema da ***satisfazibilidade*** (***de fórmulas***) em termos da linguagem SAT da maneira ilustrada a seguir. Uma instância de SAT é uma fórmula booleana f composta de

1. n variáveis booleanas: $x_1, x_2, ..., x_n$;

2. m conectivos booleanos: qualquer função booleana com uma ou duas entradas e uma saída, como $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implicação), $\leftrightarrow$ (se e somente se); e
3. parênteses. (Sem prejuízo da generalidade, supomos que não existem parênteses redundantes, isto é, a fórmula contém no máximo um par de parênteses por conetivo booleano.)

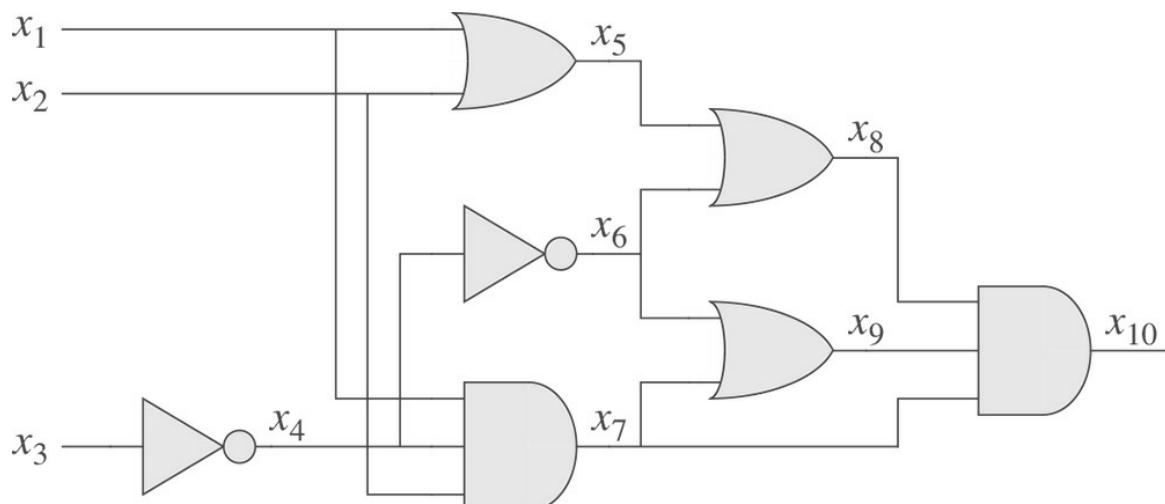
É fácil codificar uma fórmula booleana f em um comprimento que é polinomial em $n + m$. Como em circuitos combinacionais booleanos, uma ***atribuição verdade*** para uma fórmula booleana f é um conjunto de valores para as variáveis de f, e uma ***atribuição satisfatória*** é uma atribuição verdade que faz com que ela seja avaliada como 1. Uma fórmula com uma atribuição que satisfaz é uma fórmula ***satisfazível***. O problema da satisfazibilidade pergunta se uma dada fórmula booleana é satisfazível; em termos das linguagens formais,

$$\text{SAT} = \{\langle \phi \rangle : \phi \text{ é uma fórmula booleana satisfazível}\} .$$

Como exemplo, a fórmula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

tem a atribuição $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$ que a satisfaz, já que

$$\begin{aligned} \phi &= ((0 \rightarrow 0) \vee \neg ((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg (1 \vee 1) \wedge 1) \\ &= (1 \vee 0) \vee 1 \\ &= 1 , \end{aligned} \tag{34.2}$$

e, assim, essa fórmula f pertence a SAT.

O algoritmo ingênuo para determinar se uma fórmula booleana arbitrária é satisfazível não é executado em tempo polinomial. A fórmula com n tem 2_n atribuições possíveis. Se o comprimento de $\langle f\rangle$ é polinomial em n, verificar cada atribuição requer o tempo (2_n), que é superpolinomial no comprimento de $\langle f\rangle$. Como mostra o teorema a seguir, é improvável que exista um algoritmo de tempo polinomial.

Teorema 34.9

A satisfazibilidade de fórmulas booleanas é NP-completa.

Prova Começamos demonstrando que SAT $\in$ NP. Então, provaremos que SAT é NP-difícil, mostrando que CIRCUIT-SAT SAT; pelo Lema 34.8, isso provará o teorema.

Figura 34.10 Redução da satisfazibilidade de circuito à satisfazibilidade de fórmula. A fórmula produzida pelo algoritmo de redução tem uma variável para cada fio no circuito.

Para mostrar que SAT pertence a NP, mostramos que um certificado que consiste em uma atribuição satisfatória para uma fórmula de entrada *f* pode ser verificado em tempo polinomial. O algoritmo de verificação simplesmente substitui cada variável na fórmula por seu valor correspondente e depois avalia a expressão, de um modo muito semelhante ao que fizemos na equação (34.2) . Essa tarefa é fácil de realizar em tempo polinomial. Se a expressão tem o valor 1, o algoritmo verificou que a fórmula é satisfazível. Assim, a primeira condição do Lema 34.8 para a NP-completude é válida.

Para provar que SAT é NP-difícil, mostramos que CIRCUIT-SAT P SAT. Em outras palavras, precisamos mostrar como reduzir qualquer instância de satisfazibilidade de circuito a uma instância de satisfazibilidade de fórmula em tempo polinomial. Podemos usar indução para expressar qualquer circuito combinacional booleano como uma fórmula booleana. Simplesmente observamos a porta que produz a saída do circuito e expressamos indutivamente cada uma das entradas da porta como fórmulas. Então, obtemos a fórmula para o circuito escrevendo uma expressão que aplica a função da porta às fórmulas de suas entradas.

Infelizmente, esse método direto não equivale a uma redução de tempo polinomial. Como o Exercício 34.4-1 lhe pede para mostrar, subfórmulas compartilhadas — que surgem de portas cujos fios de saída têm leque de saída 2 ou maior — podem fazer o tamanho da fórmula gerada crescer exponencialmente. Assim, o algoritmo de redução deve ser um pouco mais inteligente.

A Figura 34.10 ilustra como superamos esse problema usando como exemplo o circuito da Figura 34.8(a). Para cada fio x_i no circuito *C*, a fórmula *f* tem uma variável x_i. Agora podemos expressar como uma porta funciona sob a forma de uma pequena fórmula que envolve as variáveis de seus fios incidentes. Por exemplo, o funcionamento da porta de saída AND é $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$. Denominamos cada uma dessas pequenas fórmulas por ***cláusula.***

A fórmula *f* produzida pelo algoritmo de redução é o AND da variável do circuito de saída com a conjunção de cláusulas que descrevem a operação de cada porta. Para o circuito da figura, a fórmula é

$$\begin{aligned}\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_8)\\ &\wedge (x_5 \leftrightarrow (x_1 \vee x_2))\\ &\wedge (x_6 \leftrightarrow \neg x_4)\\ &\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))\\ &\wedge (x_8 \leftrightarrow (x_5 \vee x_6))\\ &\wedge (x_9 \leftrightarrow (x_6 \vee x_7))\\ &\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) .\end{aligned}$$

Dado um circuito *C*, produzir tal fórmula *f* em tempo polinomial é direto.

Por que o circuito *C* é satisfazível exatamente quando a fórmula *f* é satisfazível? Se *C* tem atribuição que satisfaz, cada fio do circuito tem um valor bem definido, e a saída do circuito é 1. Portanto, quando atribuímos valores de fios a variáveis em *f*, cada cláusula de *f* tem valor 1 e, assim, a conjunção de todos elas tem valor 1. Ao contrário, se alguma atribuição faz *f* apresentar o valor 1, o circuito *C* é satisfazível por um argumento análogo. Assim, mostramos que CIRCUIT-SAT P SAT, o que conclui a prova.

Satisfazibilidade 3-CNF

Podemos provar que muitos problemas são NP-completos reduzindo a partir da satisfazibilidade de fórmulas. Entretanto, o algoritmo de redução deve tratar qualquer fórmula de entrada, e isso pode levar a um número enorme de casos que temos de considerar. Muitas vezes, preferimos reduzir partindo de uma linguagem restrita de fórmulas

booleanas, de modo que precisamos considerar um número menor de casos. É claro que não devemos restringir tanto a linguagem a ponto de ela se tornar resolvível em tempo polinomial. Uma linguagem conveniente é a satisfazibilidade 3-CNF, ou 3-CNF-SAT.

Definimos satisfazibilidade 3-CNF usando os termos a seguir. Um ***literal*** em uma fórmula booleana é uma ocorrência de uma variável ou sua negação. Uma fórmula booleana está em ***forma normal conjuntiva***, ou ***CNF*** (*conjunctive normal form*), se é expressa como um AND de ***cláusulas***, cada uma das quais é o OR de um ou mais literais. Uma fórmula booleana está em ***forma normal 3-conjuntiva***, ou ***3-CNF***, se cada cláusula tem exatamente três literais distintos.

Por exemplo, a fórmula booleana

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

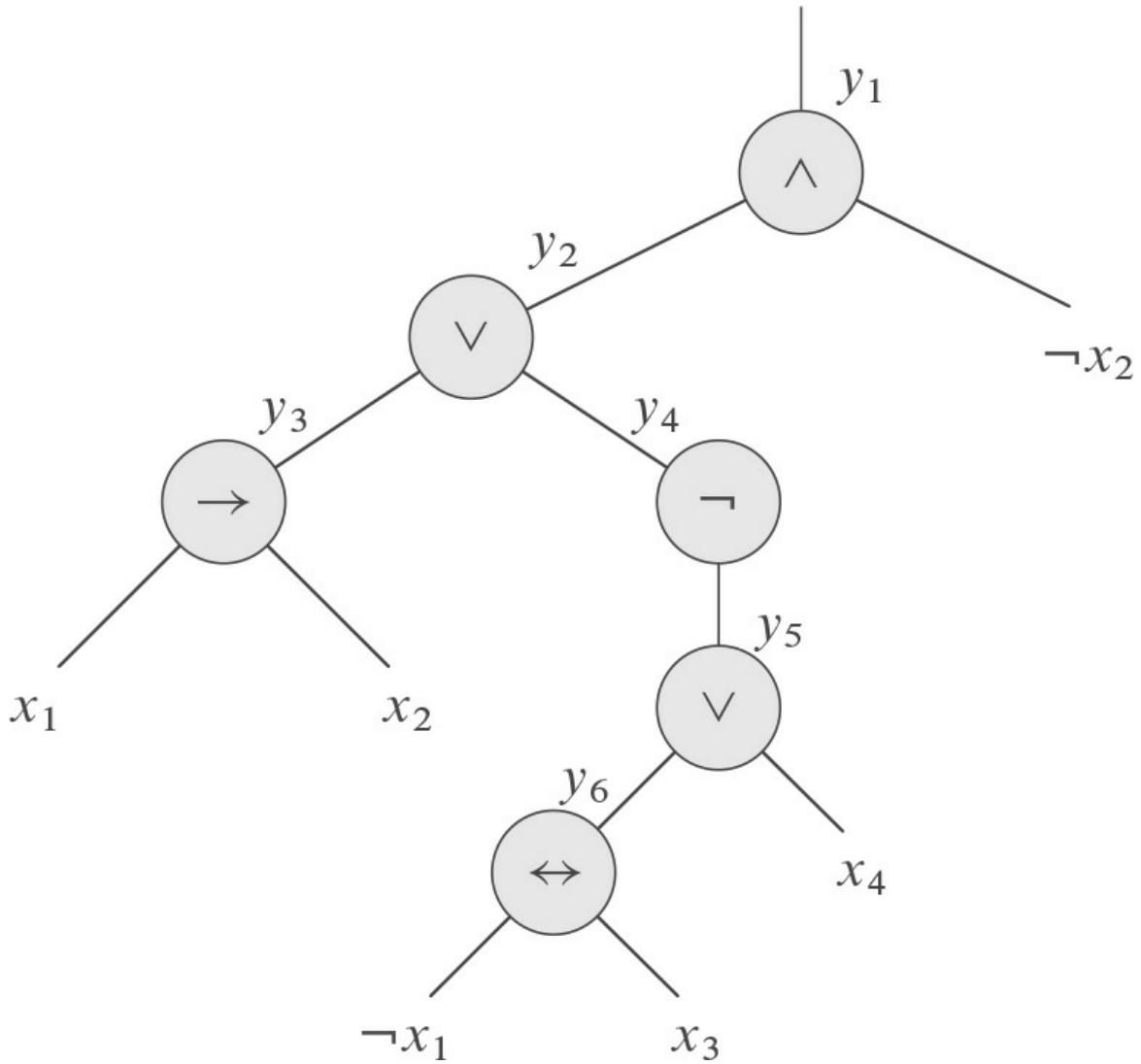
está em 3-CNF. A primeira de suas três cláusulas é $(x_1 \vee \neg x_1 \vee \neg x_2)$, que contém os três literais x_1, $\neg x_1$ e $\neg x_2$.

3-CNF-SAT pergunta se determinada fórmula booleana *f* em 3-CNF é satisfazível. O teorema a seguir mostra que é improvável que exista um algoritmo de tempo polinomial que pode determinar a satisfazibilidade de fórmulas booleanas, mesmo quando elas são expressas nessa forma normal simples.

Teorema 34.10

A satisfazibilidade de fórmulas booleanas em forma normal conjuntiva 3 é NP-completa.

Prova O argumento que usamos na prova do Teorema 34.9 para mostrar que SAT $\in$ NP se aplica igualmente bem aqui para mostrar que 3-CNF-SAT $\in$ NP. Portanto, pelo Lema 34.8, basta mostrar que SAT ^{P} 3-CNF-SAT.

Dividimos o algoritmo de redução em três etapas básicas. Cada etapa transforma progressivamente a fórmula de entrada f, deixando-a mais próxima da forma normal 3-conjuntiva desejada.

A primeira etapa é semelhante à que usamos para provar CIRCUIT-SAT ^{P} SAT no Teorema 34.9. Primeiro, construímos uma árvore binária "de análise" para a fórmula de entrada *f*, com literais como folhas e conectivos como nós internos. A Figura 34.11 mostra tal árvore de análise para a fórmula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg ((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \,. \tag{34.3}$$

Caso a fórmula de entrada contenha uma cláusula a OR de diversos literais, usamos associatividade para parentizar totalmente a expressão, de modo que cada nó interno na árvore resultante tenha um ou dois filhos. Agora podemos imaginar a árvore de análise como um circuito para calcular a função.

Imitando a redução na prova do Teorema 34.9, introduzimos uma variável y_i para a saída de cada nó interno. Em seguida, reescrevemos a fórmula original *f* como a AND da variável da raiz e uma conjunção de cláusulas que descreve o funcionamento de cada nó. Para a fórmula (34.3), a expressão resultante é

Figura 34.11 Árvore correspondente à fórmula $f = ((x_1 \rightarrow x_2) \vee \neg ((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

$$\begin{aligned}\phi = \ & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2))\\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4))\\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2))\\ & \wedge (y_4 \leftrightarrow \ \neg y_5))\\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4))\\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)).\end{aligned}$$

Observe que a fórmula f' assim obtida é uma conjunção de cláusulas f_i', cada qual com no máximo três literais. O único requisito que poderíamos deixar de cumprir é que cada cláusula seja uma OR de três literais.

A segunda etapa da redução converte cada cláusula f_i' para a forma normal conjuntiva. Construímos uma tabela verdade para f_i' avaliando todas as possíveis atribuições para suas variáveis. Cada linha da tabela verdade consiste em uma atribuição possível das variáveis da cláusula, juntamente com o valor da cláusula sob essa atribuição. Usando as entradas da tabela verdade cujo valor é 0, construímos uma fórmula em ***forma normal disjuntiva*** (ou ***DNF*** — *disjunctive normal form*) — uma OR de ANDs — que é equivalente a $\neg f_i'$. Então, negativamos essa fórmula e a convertemos em uma fórmula CNF f_i'', usando as ***leis de DeMorgan*** para lógica proposicional,

$$\neg(a \wedge b) = \neg a \vee b,$$
$$\neg(a \vee b) = \neg a \wedge b,$$

para complementar todos os literais, trocar ORs por ANDs e ANDs por OR.

Em nosso exemplo, convertemos a cláusula $f1'$ = $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ em CNF da maneira descrita a seguir. A tabela verdade para $f1'$ é dada na Figura 34.12. A fórmula DNF equivalente a $\neg f1'$ é

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \wedge (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2).$$

Negativando e aplicando as leis de DeMorgan, obtemos a fórmula CNF

$$\begin{aligned}\phi_1'' = {} & (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ & \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)\,,\end{aligned}$$

que é equivalente à cláusula original $f1'$.

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Figura 34.12 A tabela verdade para a cláusula $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

Nesse ponto, já convertemos cada cláusula fi' da fórmula f' em uma fórmula CNF fi'' e, assim, f' é equivalente à fórmula f'' que consiste na conjunção de fi''. Além disso, cada cláusula de f'' tem no máximo três literais.

A terceira e última etapa da redução prossegue na transformação da fórmula, de modo que cada cláusula tenha *exatamente* três literais distintos. Construímos a fórmula 3-CNF final f''' pelas cláusulas da fórmula CNF f''. A fórmula f

também usa duas variáveis auxiliares que denominaremos p e q. Para cada cláusula C_i de f'', incluímos as seguintes cláusulas em f''':

- Se C_i tem três literais distintos, simplesmente inclua C_i como uma cláusula de f'''.
- Se C_i tem dois literais distintos, isto é, se $C_i = (l_1 \vee l_2)$, onde l_1 e l_2 são literais, inclua $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ como cláusulas de f'''. Os literais p e $\neg p$ cumprem apenas o requisito sintático que exige que cada cláusula de f'' tenha exatamente três literais distintos. Se $p = 0$ ou $p = 1$, uma das cláusulas é equivalente a $l_1 \vee l_2$ e o valor da outra é 1, o que é a identidade para AND.
- Se C_i tem apenas um literal distinto l, inclua $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ como cláusulas de f'''. Independentemente dos valores de p e q, uma das quatro cláusulas é equivalente a l, e o valor das outras três é 1.

Podemos ver que a fórmula 3-CNF f''' é satisfazível se e somente se f é satisfazível inspecionando cada uma das três etapas. Como ocorreu na redução de CIRCUIT-SAT a SAT, a construção de f' por f na primeira etapa preserva a satisfazibilidade. A segunda etapa produz uma fórmula CNF f'' que é algebricamente equivalente a f'. A terceira etapa produz uma fórmula 3-CNF f''' que é efetivamente equivalente a f'', já que qualquer atribuição às variáveis p e q produz uma fórmula que é algebricamente equivalente a f''.

Devemos mostrar também que a redução pode ser calculada em tempo polinomial. Construir f' a partir de f introduz no máximo uma variável e uma cláusula por conectivo em f. Construir f'' a partir de f' pode introduzir no máximo oito cláusulas em f'' para cada cláusula de f', já que cada cláusula de f' tem no máximo três variáveis, e a tabela verdade para cada cláusula tem no máximo $2^3 = 8$ linhas. A construção de f''' a partir de f'' introduz no máximo quatro cláusulas em f''' para cada cláusula de f''. Portanto, o tamanho da fórmula resultante f''' é polinomial no comprimento da fórmula original. Cada uma das construções pode ser facilmente realizada em tempo polinomial.

Exercícios

34.4-1 Considere a redução direta (de tempo não polinomial) na prova do Teorema 34.9. Descreva um circuito de tamanho n que, quando convertido para uma fórmula por esse método, produz uma fórmula cujo tamanho é exponencial em n.

34.4-2 Mostre a fórmula 3-CNF que resulta quando usamos o método do Teorema 34.10 para a fórmula (34.3).

34.4-3 O professor Jagger propõe mostrar que SAT $\leq_P$ 3-CNF-SAT usando somente a técnica da tabela verdade na prova do Teorema 34.10 e não as outras etapas. Isto é, o professor propõe tomar a fórmula booleana f, formar uma tabela verdade para suas variáveis, deduzir da tabela verdade uma fórmula em 3-DNF que seja equivalente a $\neg f$ e depois negativar e aplicar as leis de DeMorgan para produzir uma fórmula 3-CNF equivalente a f. Mostre que essa estratégia não produz uma redução de tempo polinomial.

34.4-4 Mostre que o problema de determinar se uma fórmula booleana é uma tautologia é completo para co-NP. (*Sugestão*: Consulte o Exercício 34.3-7.)

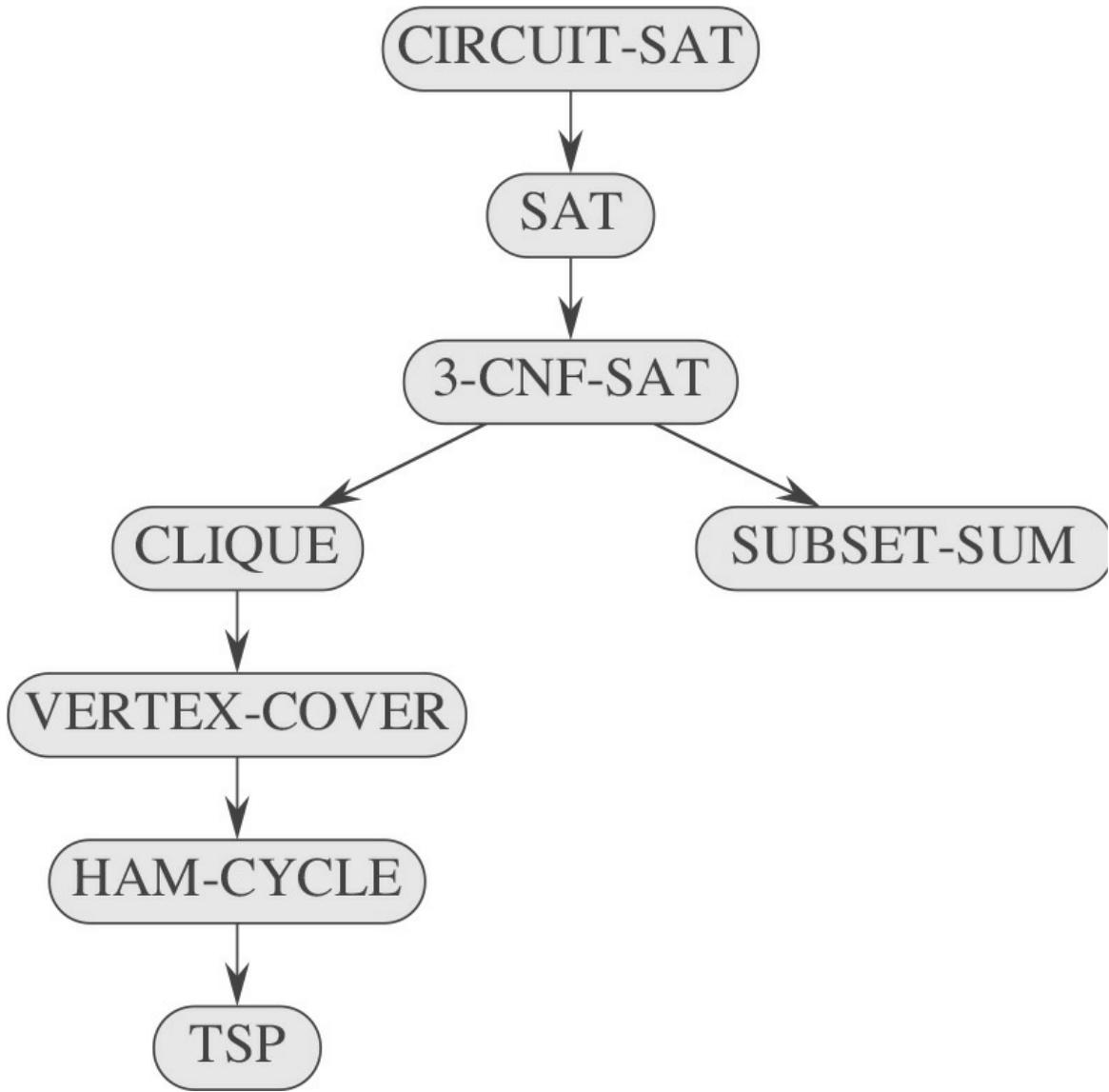
34.4-5 Mostre que o problema de determinar a satisfazibilidade de fórmulas booleanas em forma normal disjuntiva é resolvível em tempo polinomial.

34.4-6 Suponha que alguém lhe dê um algoritmo de tempo polinomial para decidir a satisfazibilidade de fórmulas. Descreva como usar esse algoritmo para encontrar atribuições que a satisfazem em tempo polinomial.

34.4-7 Seja 2-CNF-SAT o conjunto de fórmulas booleanas satisfazíveis em CNF com exatamente dois literais por cláusula. Mostre que 2-CNF-SAT $\in$ P. O seu algoritmo deve ser o mais eficiente possível. (*Sugestão*: Observe que $x \vee y$ é equivalente a $\neg x \rightarrow y$. Reduza 2-CNF-SAT a um problema eficientemente resolvível em um grafo dirigido.)

34.5 PROBLEMAS NP-COMPLETOS

Os problemas NP-completos surgem em diversos domínios: lógica booleana, grafos, aritmética, projeto de rede, conjuntos e partições, armazenamento e recuperação, sequenciamento e escalonamento, programação matemática, álgebra e teoria dos números, jogos e quebra-cabeças, autômatos e teoria das linguagens, otimização de programas, biologia, química, física e outros. Nesta seção, usaremos a metodologia de redução para dar provas de NP-completude para uma variedade de problemas extraídos da teoria dos grafos e do particionamento de conjuntos.

A Figura 34.13 representa a estrutura das provas de NP-completude nesta seção e na Seção 34.4. Provamos que cada linguagem na figura é NP-completa por redução da linguagem que aponta para ela. Na raiz está CIRCUIT-SAT, que provamos ser NP-completo no Teorema 34.7.

34.5.1 O PROBLEMA DO CLIQUE

Um ***clique*** em um grafo não dirigido $G = (V, E)$ é um subconjunto $V' \subseteq V$ de vértices, no qual cada par está ligado por uma aresta em E. Em outras palavras, um clique é um subgrafo completo de G. O ***tamanho*** de um clique é o número de vértices que contém. O ***problema do clique*** é o problema de otimização de encontrar um clique de tamanho máximo em um grafo. Por ser um problema de decisão, simplesmente perguntamos se um clique de um dado tamanho k existe no grafo. A definição formal é

$$\text{CLIQUE} = \{\langle G, K \rangle : G \text{ é um grafo com um clique de tamanho } k\} .$$

Um algoritmo ingênuo para determinar se um grafo $G = (V, E)$ com $|V|$ vértices tem um clique de tamanho k é fazer uma lista de todos os subconjuntos k de V e conferir cada um para ver se ele forma um clique. O tempo de execução desse algoritmo é $(k_2(|kV)|)$, que é polinomial se k é uma constante. Porém, em geral k poderia estar próximo de $|V|/2$ e, nesse caso, o algoritmo é executado em tempo superpolinomial. Na verdade, é improvável que exista um algoritmo eficiente para o problema do clique.

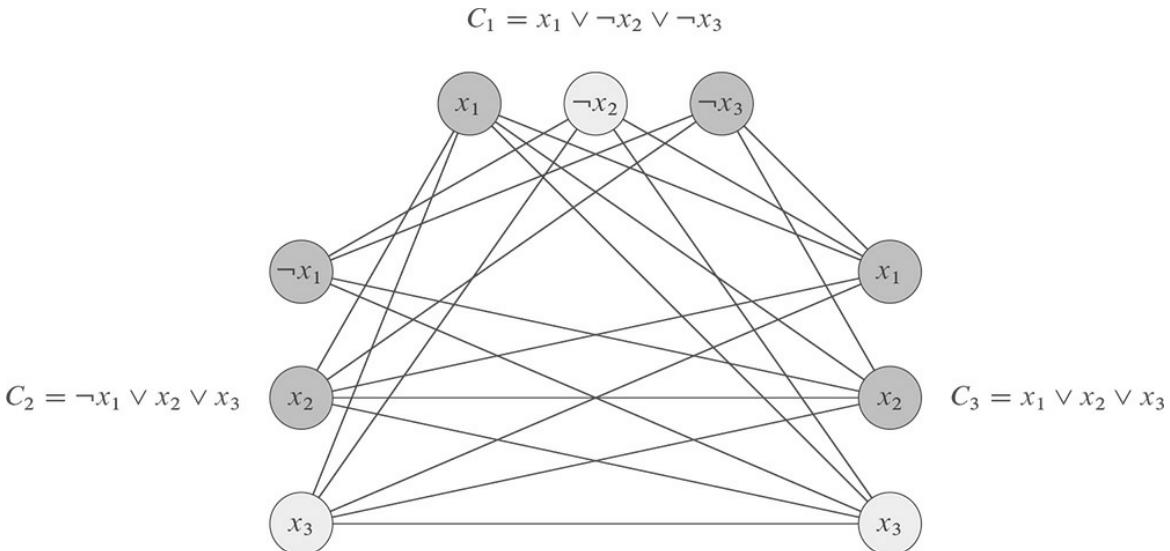
Figura 34.13 Estrutura de provas de NP-completude nas Seções 34.4 e 34.5. Todas as provas decorrem em última análise por redução da NP-completude de CIRCUIT-SAT.

Teorema 34.11

O problema do clique é NP-completo.

Prova Para mostrar que CLIQUE ∈ NP, para um dado grafo $G = (V, E)$, usamos o conjunto $V' \subseteq V$ de vértices no clique como um certificado para G. Podemos verificar se V' é um clique em tempo polinomial verificando se, para cada par $u, v \in V'$, a aresta (u, v) pertence a E.

Em seguida, provamos que 3-CNF-SAT $\leq_P$ CLIQUE, o que mostra que o problema do clique é NP-difícil. É surpreendente que possamos provar tal resultado, já que, à primeira vista, fórmulas lógicas parecem ter pouco a ver com grafos.

O algoritmo de redução começa com uma instância de 3-CNF-SAT. Seja $f = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ uma fórmula booleana em 3-CNF com k cláusulas. Para $r = 1, 2, ..., k$, cada cláusula C_r tem exatamente três literais distintos l_{r1}, l_{r2}, e l_{r3}. Construiremos um grafo G tal que f seja satisfazível se e somente se G tem um clique de tamanho k.

Construiremos o grafo $G = (V, E)$ da seguinte maneira. Para cada cláusula $C_r = l^{r1} \vee l^{r2} \vee l^{r3})$ em f, inserimos uma tripla de vértices v_{r1}, v_{r2} e v_{r3} em V. Inserimos uma aresta entre dois vértices $v^{\ i}$ e v^{sj} se ambas as afirmativas seguintes valem:

- v_r^i e v_s^j estão em triplas diferentes, isto é, $r \neq s$ e
- seus literais correspondentes são ***coerentes***, isto é, l_r^i não é a negação de l_s^j.

É fácil construir esse grafo a partir de f em tempo polinomial. Como exemplo dessa construção, se temos

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) ,$$

então G é o grafo mostrado na Figura 34.14.

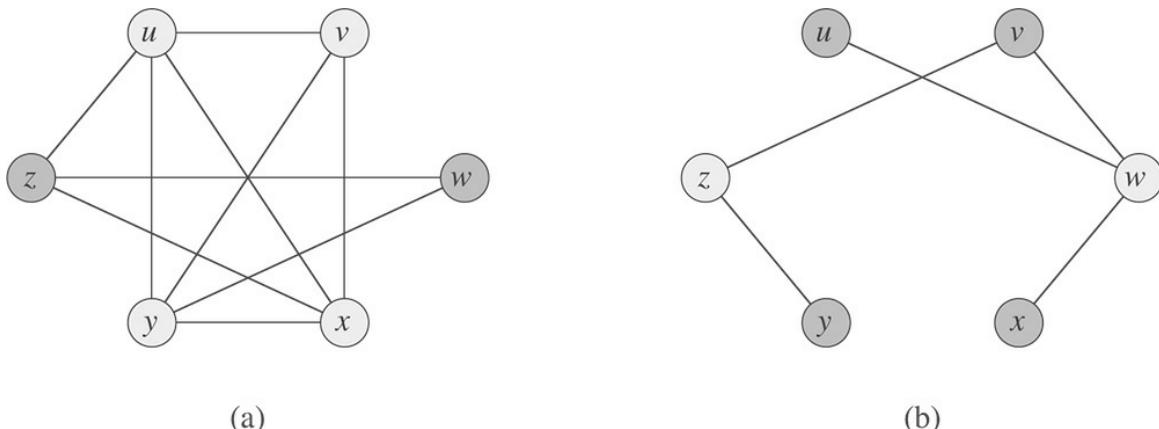
Devemos mostrar que essa transformação de f em G é uma redução. Primeiro, suponha que f tenha uma atribuição que satisfaz. Então, cada cláusula C_r contém no mínimo um literal l_r^i ao qual é atribuído 1, e tal literal corresponde a um vértice v_r^i. Escolher um desses literais "verdadeiros" de cada cláusula produz um conjunto de V' de k vértices. Afirmamos que V' é um clique. Para quaisquer dois vértices $v_r^i, v_s^j \in V'$, onde $r \neq s$, ambos os literais correspondentes, l_r^i e l_s^j, mapeiam para 1 pela atribuição dada e, portanto, os literais não podem ser complementos. Assim, pela construção de G, a aresta (v_r^i, v_s^j) pertence a E.

Figura 34.14 O grafo G derivado da fórmula 3-CNF $f = C_1 \wedge C_2 \wedge C_3$, onde $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$ e $C_3 = (x_1 \vee x_2 \vee x_3)$ na redução de 3-CNF-SAT a CLIQUE. Uma atribuição que satisfaz a fórmula tem $x_2 = 0$, $x_3 = 1$, e x_1 pode ser 0 ou 1. Essa atribuição satisfaz C_1 com $\neg x_2$ e C_2 e C_3 com x_3, correspondente ao clique com vértices sombreados em tom mais claro.

Inversamente, suponha que G tenha uma clique V' de tamanho k. Nenhuma aresta em G liga vértices na mesma tripla e, portanto, V' contém exatamente um vértice por tripla. Podemos atribuir 1 a cada literal l_r^i tal que $v_r^i \in V'$, sem receio de atribuir 1 a um literal e a seu complemento, já que G não contém arestas entre literais incoerentes. Cada cláusula é satisfeita, e portanto f é satisfeita. (Quaisquer variáveis que não correspondam a nenhum vértice no clique podem ser definidas arbitrariamente.

No exemplo da Figura 34.14, uma atribuição que satisfaz de f tem $x_2 = 0$ e $x_3 = 1$. Um clique correspondente de tamanho $k = 3$ consiste nos vértices que correspondem a $\neg x_2$ da primeira cláusula, x_3 da segunda cláusula e x_3 da terceira cláusula. Como o clique não contém nenhum vértice correspondente a x_1 nem a $\neg x_1$, podemos definir x_1 como 0 ou 1 nessa atribuição satisfatória.

Observe que, na prova do Teorema 34.11, reduzimos uma instância arbitrária de 3-CNF-SAT a uma instância de CLIQUE com uma estrutura específica. Pode parecer que mostramos apenas que CLIQUE é NP-difícil em grafos nos quais os vértices estão restritos a ocorrer em triplas e nos quais não há arestas entre vértices na mesma tripla. Realmente, já mostramos que CLIQUE é NP-difícil apenas nesse caso restrito, mas essa prova basta para mostrar que

CLIQUE é NP-difícil em grafos gerais. Por quê? Se tivéssemos um algoritmo de tempo polinomial que resolvesse CLIQUE em grafos gerais, ele também resolveria CLIQUE em grafos restritos.

Todavia, a abordagem inversa — reduzir instâncias de 3-CNF-SAT com uma estrutura especial a instâncias gerais de CLIQUE — também não teria sido suficiente. Por quê? Talvez as instâncias de 3-CNF-SAT que escolhemos para iniciar a redução fossem "fáceis", e portanto não teríamos reduzido um problema NP-difícil a CLIQUE.

Observe também que a redução usou a instância de 3-CNF-SAT, mas não a solução. Teríamos errado se a redução de tempo polinomial tivesse sido baseada em saber se a fórmula f é satisfazível, já que não sabemos como decidir se f é satisfazível em tempo polinomial.

34.5.2 O PROBLEMA DE COBERTURA DE VÉRTICES

Uma ***cobertura de vértices*** de um grafo não dirigido $G = (V, E)$ é um subconjunto $V' \subseteq V$ tal que $(u, v) \in E$, então $u \in V'$ ou $v \in V'$ (ou ambos). Isto é, cada vértice "cobre" suas arestas incidentes, e uma cobertura de vértices para G é um conjunto de vértices que cobre todas as arestas em E. O ***tamanho*** de uma cobertura de vértices é o número de vértices que contém. Por exemplo, o grafo na Figura 34.15(b) tem uma cobertura de vértices $\{w, z\}$ de tamanho 2.

O problema de cobertura de vértices é o de encontrar uma cobertura de vértices de tamanho mínimo em dado grafo. Enunciando novamente esse problema de otimização como um problema de decisão, desejamos determinar se um grafo tem uma cobertura de vértices de tamanho k dado. Como linguagem, definimos

Figura 34.15 Redução CLIQUE a VERTEX-COVER. **(a)** Um grafo não dirigido $G = (V, E)$ com clique $V' = \{u, v, x, y\}$. **(b)** O grafo G produzido pelo algoritmo de redução que tem cobertura de vértices $V - V' = \{w, z\}$.

VERTEX-COVER = $\{\langle G, k\rangle$: grafo G tem uma cobertura de vértices de tamanho $k\}$.

O teorema a seguir mostra que esse problema é NP-completo.

Teorema 34.12

O problema de cobertura de vértices é NP-completo.

Prova Primeiro mostramos que VERTEX-COVER $\in$ NP. Vamos supor que tenhamos um grafo $G = (V, E)$ e um inteiro k. O certificado que escolhemos é a própria cobertura de vértices $V' \subseteq V$. O algoritmo de verificação afirma que $|V'| = k$, e então verifica, para cada aresta $(u, v) \in E$, que $u \in V'$ ou $v \in V'$. É fácil verificar o certificado em tempo polinomial.

Provamos que o problema de cobertura de vértices é NP-difícil mostrando que CLIQUE $\leq_P$ VERTEX-COVER. Essa redução se baseia na noção de "complemento" de um grafo. Dado um grafo não dirigido $G = (V, E)$, definimos o *complemento* de G como $G = (V, E)$, onde $E = \{(u, v) : u, v \in V, u \neq v$ e $(u, v) \notin E\}$. Em outras palavras, G é o grafo que contém exatamente as arestas que não estão em G. A Figura 34.15 mostra um grafo e seu complemento, e ilustra a redução de CLIQUE a VERTEX-COVER.

O algoritmo de redução toma como entrada uma instância $\langle G, k\rangle$ do problema do clique. Calcula o complemento G, o que é fácil de fazer em tempo polinomial. A saída do algoritmo de redução é a instância $\langle G, |V| - k\rangle$ do problema de cobertura de vértices. Para concluir a prova, mostramos que essa transformação é de fato uma redução: o grafo G tem um clique de tamanho k se e somente se o grafo tem uma cobertura de vértices de tamanho $|V| - k$.

Suponha que G tenha um clique $V' \subseteq V$ com $|V'| = k$. Afirmamos que $V - V'$ é uma cobertura de vértices em G. Seja (u, v) qualquer aresta em E. Então, $(u, v) \notin E$, o que implica que pelo menos um de u e v não pertence a V', já que todo par de vértices em V' está ligado por uma aresta de E. De modo equivalente, pelo menos um de u e v está em $V - V'$, o que significa que a aresta (u, v) é coberta por $V - V'$. Visto que (u, v) foi escolhida arbitrariamente em E, toda aresta de E é coberta por um vértice em $V - V'$. Consequentemente, o conjunto $V - V'$, que tem tamanho $|V| - k$, forma uma cobertura de vértices para G.

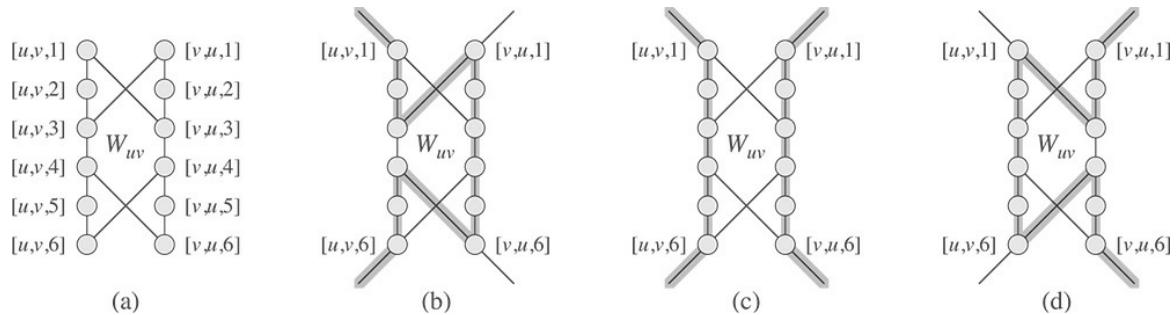
Ao contrário, suponha que G tenha uma cobertura de vértices $V' \subseteq V$, onde $|V'| = |V| - k$. Então, para todo $u, v \in V$, se $(u, v) \in E$, então $u \in V'$ ou $v \in V'$, ou ambos. A contrapositiva dessa implicação é que, para todo $u, v \in V$, se $u \notin V'$ e $v \notin V'$, então $(u, v) \in E$. Em outras palavras, $V - V'$ é um clique e tem tamanho $|V| - |V'| = k$.

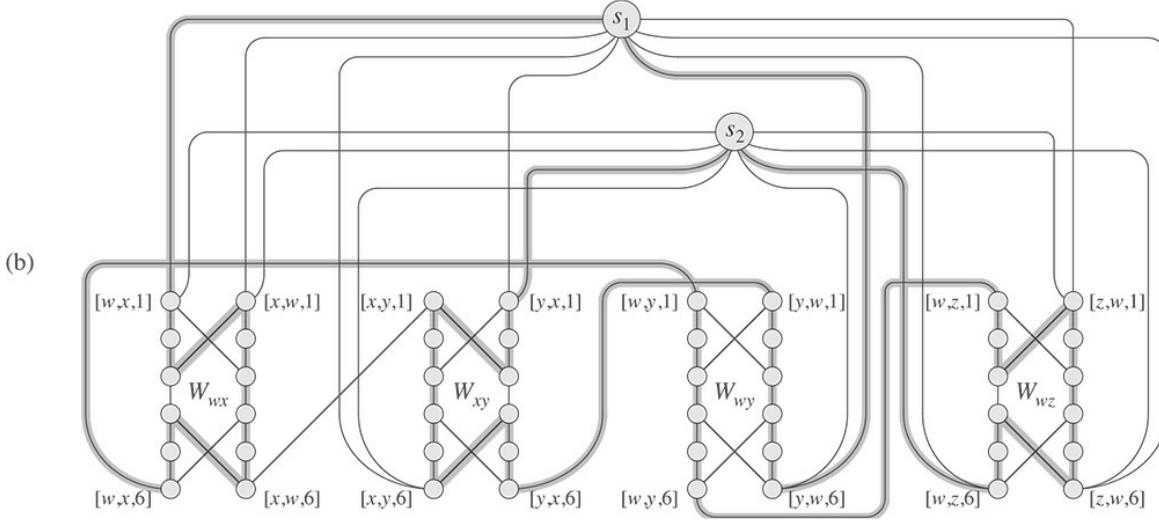
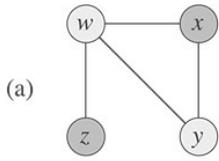
Visto que VERTEX-COVER é NP-completo, não esperamos encontrar um algoritmo de tempo polinomial para determinar uma cobertura de vértices de tamanho mínimo. Contudo, a Seção 35.1 apresenta um "algoritmo de aproximação" de tempo polinomial, que produz soluções "aproximadas" para o problema de cobertura de vértices. O tamanho de uma cobertura de vértices produzida pelo algoritmo é no máximo duas vezes o tamanho mínimo de uma cobertura de vértices. Assim, não devemos deixar de ter esperança só porque um problema é NP-completo. Talvez possamos projetar um algoritmo de aproximação de tempo polinomial que obtenha soluções aproximadas, embora descobrir uma solução ótima seja NP-completo. O Capítulo 35 dá vários algoritmos de aproximação para NP-completo.

34.5.3 O PROBLEMA DO CICLO HAMILTONIANO

Voltamos agora ao problema do ciclo hamiltoniano definido na Seção 34.2.

Teorema 34.13

O problema do ciclo hamiltoniano é NP-completo.

Prova Primeiro mostramos que HAM-CYCLE pertence a NP. Dado um grafo $G = (V, E)$, nosso certificado é a sequência de vértices $|V|$ que forma o ciclo hamiltoniano. O algoritmo de verificação confere se essa sequência contém cada vértice em V exatamente uma vez e se, com o primeiro vértice repetido no final, ela forma um ciclo em G. Isto é, verifica se existe uma aresta entre cada par de vértices consecutivos e entre o primeiro e o último vértices. Podemos verificar o certificado em tempo polinomial.

Agora provamos que VERTEX-COVER $\leq_P$ HAM-CYCLE, o que mostra que HAM-CYCLE é NP-completo. Dado um grafo não dirigido $G = (V, E)$ e um inteiro k, construímos um grafo não dirigido $G' = (V', E')$ que tem um ciclo hamiltoniano se e somente se G tem uma cobertura de vértices de tamanho k.

Nossa construção é baseada em um ***widget***, que é um fragmento de um grafo que impõe certas propriedades. A Figura 34.16(a) mostra o widget que usamos. Para cada aresta $(u, v) \in E$, o grafo G' que construímos conterá uma cópia desse widget, que denotamos por W_{uv}. Denotamos cada vértice em W_{uv} por $[u, v, i]$ ou $[v, u, i]$, onde $1 \leq i \leq 6$, de modo que cada widget W_{uv} contém 12 vértices. O widget W_{uv} também contém as 14 arestas mostradas na Figura

34.16(a). Junto com a estrutura interna do widget, impomos as propriedades que queremos, limitando as conexões entre o widget e o restante do grafo G' que construímos. Em particular, somente os vértices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ e $[v, u, 6]$ terão arestas incidentes que vêm de fora de W_{uv}. Qualquer ciclo hamiltoniano de G' terá de percorrer as arestas de W_{uv} em um dos três modos mostrados nas Figuras 34.16(b)–(d). Se o ciclo entrar pelo vértice $[u, v, 1]$, deve sair pelo vértice $[u, v, 6]$ e visitar todos os 12 vértices do widget (Figura 34.16(b)) ou os seis vértices de $[u, v, 1]$ a $[u, v, 6]$ (Figura 34.16(c)). Nesse último caso, o ciclo terá de entrar novamente no widget para visitar os vértices $[v, u, 1]$ a $[v, u, 6]$. De modo semelhante, se o ciclo entrar pelo vértice $[v, u, 1]$, deverá sair pelo vértice $[v, u, 6]$ e visitar todos os 12 vértices do widget (Figura 34.16(d)) ou os seis vértices de $[v, u, 1]$ a $[v, u, 6]$ (Figura 34.16(c)). Não é possível nenhum outro caminho que passe pelo widget e visite todos os 12 vértices. Em particular, é impossível construir dois caminhos disjuntos nos vértices, um dos quais ligue $[u, v, 1]$ a $[v, u, 6]$ e o outro ligue $[v, u, 1]$ a $[u, v, 6]$, tais que a união dos dois caminhos contenha todos os vértices do widget.

Os únicos vértices em V' além dos vértices dos widgets são ***vértices seletores*** s_1 , s_2 ,..., s_k. Usamos arestas incidentes em vértices seletores de G' para selecionar os k vértices da cobertura em G. Além das arestas em widgets, E' contém dois outros tipos de arestas que a Figura 34.17 mostra. Primeiro, para cada vértice $u \in V$, adicionamos arestas para unir pares de widgets de modo a formar um caminho que contém todos os widgets correspondentes a arestas incidentes em u em G. Ordenamos arbitrariamente os vértices adjacentes a cada vértice $u \in V$ como $u_{(1)}$, $u_{(2)}$,..., $u_{(\text{grau}(u))}$, onde (grau(u)) é o número de vértices adjacentes a u. Criamos um caminho em G' que passa por todos os widgets que correspondem a arestas incidentes em u adicionando a E' as arestas $\{([u, u_{(i)}, 6], [u, u_{(i+1)}, 1]) : 1 \leq i \leq \text{grau}(u) - 1\}$. Por exemplo, na Figura 34.17, ordenamos os vértices adjacentes a w como x , y , z , e assim o grafo G' da parte (b) da figura inclui as arestas $([w, x, 6], [w, y, 1])$ e $([w, y, 6], [w, z, 1])$. Para cada vértice $u \in V$, essas arestas em G' completam um caminho que contém todos os widgets que correspondem a arestas incidentes para u em G.

Figura 34.16 O widget usado para reduzir o problema de cobertura de vértices ao problema de ciclo hamiltoniano. Uma aresta (u, v) do grafo G corresponde ao widget W_{uv} no grafo G' criado na redução. **(a)** O widget, com vértices individuais identificados. **(b)–(d)** Os caminhos sombreados são os únicos possíveis que passam pelo widget e incluem todos os vértices, considerando que as únicas ligações do widget com o restante de G' são realizadas pelos vértices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ e $[v, u, 6]$.

Figura 34.17 Redução de uma instância do problema da cobertura de vértices a uma instância do problema do ciclo hamiltoniano. **(a)** Um grafo não dirigido G com uma cobertura de vértices de tamanho 2, que consiste nos vértices sombreados em tom mais claro w e y. **(b)** O grafo não dirigido G' produzido pela redução, com o caminho hamiltoniano correspondendo à cobertura de vértices sombreado. A cobertura de vértices $\{w, y\}$ corresponde às arestas $(s_1, [w, x, 1])$ e $(s_2, [y, x, 1])$ que aparecem no ciclo hamiltoniano.

A intuição por trás dessas arestas é que, se escolhermos um vértice $u \in V$ na cobertura de vértices de G, podemos construir um caminho de $[u, u_{(1)}, 1]$ até $[u, u_{(\text{grau}(u))}, 6]$ em G' que "cobre" todos os widgets que correspondem a arestas incidentes em u. Isto é, para cada um desses widgets, digamos $W_{u,u(i)}$, o caminho inclui todos os 12 vértices (se u está na cobertura de vértices, mas $u_{(i)}$ não está) ou apenas os seis vértices $[u, u_{(i)}, 1]$, $[u, u_{(i)}, 2]$, ..., $[u, u_{(i)}, 6]$ (se u e $u_{(i)}$ estão ambos na cobertura de vértices).

O último tipo de aresta em E' une o primeiro vértice $[u, u_{(1)}, 1]$ e o último vértice $[u, u_{(\text{grau}(u))}, 6]$ de cada um desses caminhos a cada um dos vértices seletores. Isto é, incluímos as arestas

$$\{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ e } 1 \le j \le k\}$$
$$\cup \{(s_j, [u, u^{(\text{grau}(u))}, 6]) : u \in V \text{ e } 1 \le j \le k\} .$$

Em seguida, mostramos que o tamanho de G' é polinomial no tamanho de G e, consequentemente, podemos construir G' em tempo polinomial no tamanho de G. Os vértices de G' são os dos widgets, mais os vértices seletores. Com 12 vértices por widget, mais $k \le |V|$ vértices seletores, temos um total de

$$\begin{aligned} |V'| &= 12\,|E| + k \\ &\le 12\,|E| + |V| \end{aligned}$$

vértices. As arestas de G' são as dos widgets, as que ficam entre widgets e as que ligam vértices seletores a widgets. Cada widget contém 14 arestas, o que dá um total de 14 $|E|$ em todos os widgets. Para cada vértice $u \in V$, o grafo G' tem arestas de grau $(u) - 1$ entre widgets. Portanto, a soma das arestas em todos os vértices em V, é

$$\sum_{u \in V} (\text{grau}(u) - 1) = 2 \, | \, E \, | - | \, V \, |$$

arestas entre widgets. Finalmente, G' tem duas arestas para cada par que consiste em um vértice seletor e um vértice de V, totalizando $2k\ |V|$ dessas arestas. O número total de arestas de G' é então

$$\begin{aligned} |E'| &= (14\ |E|) + (2\ |E| - |V|) + (2k\ |V|) \\ &= 16\ |E| + (2k - 1)\ |V| \\ &\leq 16\ |E| + (2\ |V| - 1)\ |V|\ . \end{aligned}$$

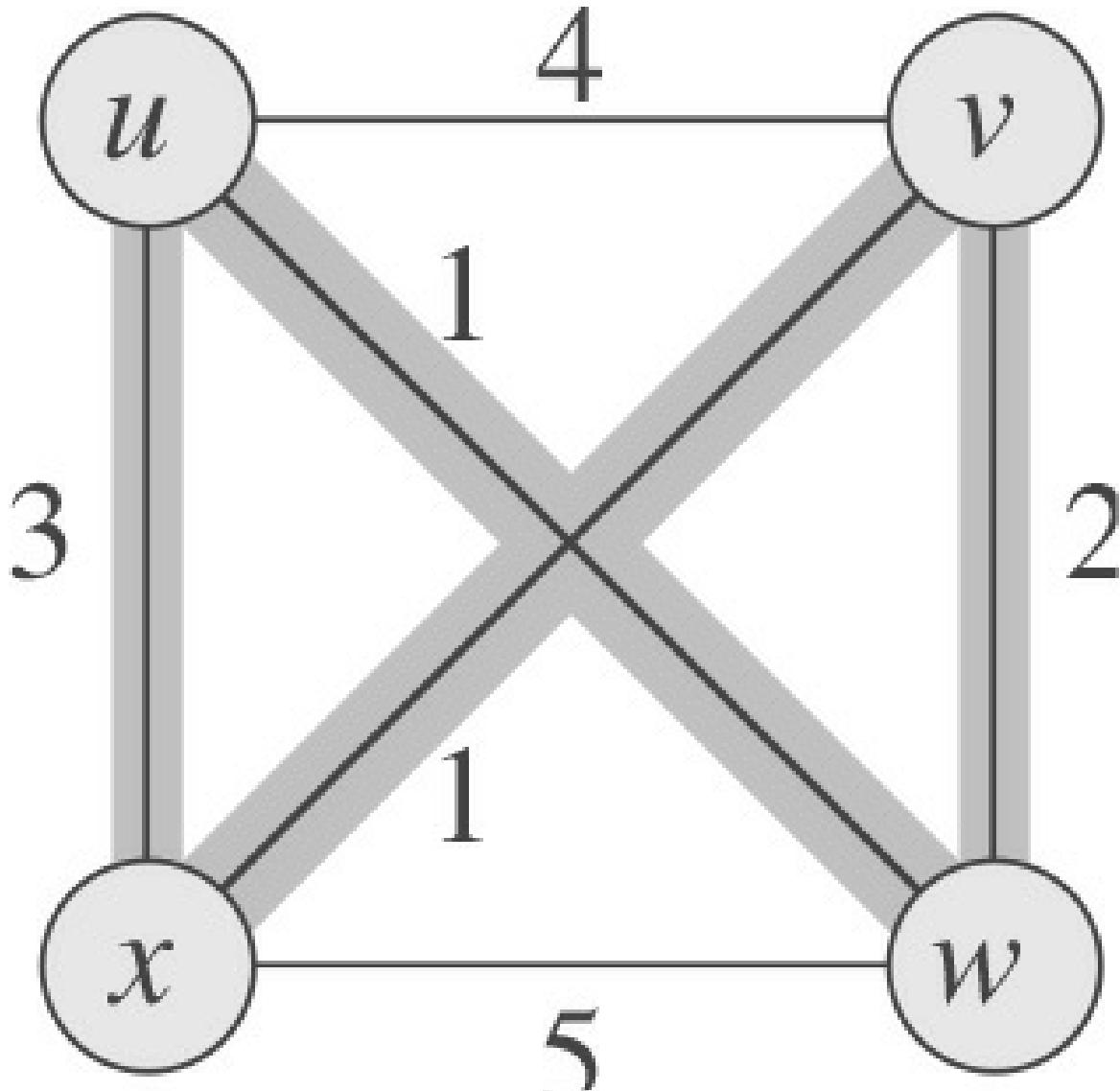
Agora, mostramos que a transformação do grafo G em G' é uma redução. Isto é, devemos mostrar que G tem uma cobertura de vértices de tamanho k se e somente se G' tem um ciclo hamiltoniano.

Suponha que $G = (V, E)$ tenha uma cobertura de vértices $V^* \subseteq V$ de tamanho k. Seja $V^* = \{u_1, u_2, ..., u_k\}$. Como mostra a Figura 34.17, formamos um ciclo hamiltoniano em G' incluindo as seguintes arestas[10] para cada vértice $u_j \in V^*$. Incluímos as arestas $\{([u_j, u_{(}^{i}), 6], [u_j, u_j^{(i+1)}, 1]) : 1 \leq i \leq \text{grau}(u_j) - 1\}$, que ligam todos os widgets que correspondem a arestas incidentes em u_j. Também incluímos as arestas contidas nesses widgets, como mostram as Figuras 34.16(b)–(d), dependendo de a aresta ser coberta por um ou dois vértices em V^*. O ciclo hamiltoniano também inclui as arestas

$$\begin{aligned} &\{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \leq j \leq k\} \\ &\qquad \cup \{(s_j + {}_1, [u_j, u_j^{(\text{grau}(u_j))}, 6]) : 1 \leq j \leq k - 1\} \\ &\qquad \cup \{(s_1, [u_k, u_k^{(\text{grau}(u_k))}, 6])\}\ . \end{aligned}$$

Examinando a Figura 34.17, pode-se verificar que essas arestas formam um ciclo. O ciclo começa em s_1, visita todos os widgets que correspondem a arestas incidentes em u_1, depois visita s_2, visita todos os widgets que correspondem a arestas incidentes em u_2, e assim por diante, até retornar a s_1. O ciclo visita cada widget uma ou duas vezes, dependendo de um ou dois vértices de V^* cobrir(em) sua aresta correspondente. Como V^* é uma cobertura de vértices para G, cada aresta em E é incidente em algum vértice de V^*, e portanto o ciclo visita cada vértice em cada widget de G'. Visto que o ciclo também visita todo vértice seletor, ele é hamiltoniano.

Inversamente, suponha que $G' = (V', E')$ tenha um ciclo hamiltoniano $C \subseteq E'$. Afirmamos que o conjunto

$$V^* = (u \in V : (s_j, [u, u^{(1)}, 1]) \in C \text{ para algum } 1 \leq j \leq k\} \tag{34.4}$$

é uma cobertura de vértices para G. Para ver por que, particione C em caminhos maximais que começam em algum vértice seletor s_i, percorrem uma aresta $(s_i, [u, u_{(1),}\ 1])$ para algum $u \in V$ e terminam em um vértice seletor s_j sem passar por nenhum outro vértice seletor. Vamos denominar cada caminho "caminho de cobertura". Dependendo de como G' é construído, cada caminho de cobertura deve começar em algum s_i, tomar a aresta $(s_i, [u, u_{(1)}\ , 1])$ correspondente a algum vértice $u \in V$, passar por todos os widgets que correspondem a arestas de E incidentes em u e depois terminar em algum vértice seletor s_j. Referimo-nos a esse caminho de cobertura como p_u e, pela equação (34.4), inserimos u em V^*. Cada widget visitado por p_u deve ser W_{uv} ou W_{vu} para algum $v \in V$. Para cada widget visitado por p_u, seus vértices são visitados por um ou por dois caminhos de cobertura. Se forem visitados por um caminho de cobertura, a aresta $(u, v) \in E$ é coberta em G pelo vértice u. Se dois caminhos de cobertura visitam o widget, o outro caminho de cobertura deve ser p_v, o que implica que $v \in V^*$, e a aresta $(u, v) \in E$ é coberta por u e por v. Como cada vértice em cada widget é visitado por algum caminho de cobertura, vemos que cada aresta em E é coberta por algum vértice em V^*.

34.5.4 O PROBLEMA DO CAIXEIRO-VIAJANTE

No ***problema do caixeiro-viajante***, que está intimamente relacionado ao problema do ciclo hamiltoniano, um vendedor deve visitar n cidades. Modelando o problema como um grafo completo com n vértices, podemos dizer que o vendedor deseja fazer um ***percurso***, ou um ciclo hamiltoniano, visitando cada cidade exatamente uma vez e terminando na cidade de onde partiu. O vendedor incorre em um custo inteiro não negativo $c(i, j)$ para viajar da cidade i para a cidade j, e deseja fazer o percurso cujo custo total seja mínimo, em que o custo total é a soma dos custos individuais ao longo das arestas do percurso. Por exemplo, na Figura 34.18, um percurso de custo mínimo é $\langle u, w, v, x, u\rangle$, com custo 7. A linguagem formal para o problema de decisão correspondente é

$$\begin{aligned}
\text{TSP} = \{\langle G, c, k\rangle : \; & G = (V, E) \text{ é um grafo completo,} \\
& c \text{ é uma função de } V \times V \to \mathbb{N}, \\
& k \in \mathbb{N} \text{ e} \\
& G \text{ tem um percurso de caixeiro-viajante com custo menor ou igual a } k\} \, .
\end{aligned}$$

Figura 34.18 Uma instância do problema do caixeiro-viajante. As arestas sombreadas representam um percurso de custo mínimo, com custo 7.

O teorema a seguir mostra que é improvável que exista um algoritmo rápido para o problema do caixeiro-viajante.

Teorema 34.14

O problema do caixeiro-viajante é NP-completo.

Prova Primeiro mostramos que TSP pertence a NP. Dada uma instância do problema, usamos como certificado a sequência de n vértices no percurso. O algoritmo de verificação confirma que essa sequência contém cada vértice exatamente uma vez, soma os custos de arestas e verifica se a soma é no máximo k. Esse processo pode certamente ser feito em tempo polinomial.

Para provar que TSP é NP-difícil, mostramos que HAM-CYCLE $\leq_P$ TSP. Seja $G = (V, E)$ uma instância de HAM-CYCLE. Construímos uma instância de TSP da maneira seguinte. Formamos o grafo completo $G' = (V, E')$, onde $E' = \{(i, j) : i, j \in V\}$ e $i \neq j\}$ e definimos a função custo c como

$$c(i,j) = \begin{cases} 0 & \text{se } (i,j) \in E, \\ 1 & \text{se } (i,j) \notin E \end{cases}$$

(Observe que, como G é não dirigido, não tem nenhum laço, assim $c(v, v) = 1$ para todos os vértices $v \in V$.) A instância de TSP é então $(G', c, 0)$, que podemos facilmente criar em tempo polinomial.

Agora, mostraremos que o grafo G tem um ciclo hamiltoniano se e somente se o grafo G' tem um percurso cujo custo é menor ou igual a 0. Suponha que o grafo G tenha um ciclo hamiltoniano h. Cada aresta em h pertence a E e, portanto, tem custo 0 em G'. Assim, h é um percurso em G' com custo 0. Ao contrário, suponha que o grafo G' tenha um percurso h' de custo menor ou igual a 0. Visto que os custos das arestas em E' são 0 e 1, o custo da percurso h' é exatamente 0, e cada aresta no percurso deve ter custo 0. Portanto, h' contém apenas arestas em E. Concluímos que h é um ciclo hamiltoniano no grafo G.

34.5.5 O PROBLEMA DA SOMA DE SUBCONJUNTOS

O próximo problema NP-completo que consideraremos é aritmético. No ***problema da soma de subconjuntos***, temos um conjunto finito S de inteiros positivos e um inteiro ***alvo*** $t > 0$. Perguntamos se existe um subconjunto $S' \subseteq S$ cuja soma de seus elementos é t. Por exemplo, se $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ e $t = 138457$, então o subconjunto $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$ é uma solução.

Como sempre, definimos o problema como uma linguagem:

$$\text{SUBSET-SUM} = \{\langle S, t\rangle : \text{existe um subconjunto } S' \subseteq S \text{ tal que } t = \textstyle\sum_{s \in S'} s\}.$$

Como ocorre com qualquer problema de aritmética, é importante lembrar que nossa codificação padrão pressupõe que os inteiros da entrada estão codificados em binário. Com isso em mente, podemos mostrar que é improvável que o problema da soma de subconjuntos tenha um algoritmo rápido.

Teorema 34.15

O problema da soma de subconjuntos é NP-completo.

Prova Para mostrar que SUBSET-SUM está em NP, para uma instância $\langle S, t\rangle$ do problema é o subconjunto S' é o certificado. Um algoritmo de verificação pode verificar se $t = \sum_{s \in S'} s$ em tempo polinomial.

Agora mostraremos que 3-CNF-SAT $\leq_P$ SUBSET-SUM. Dada uma fórmula 3-CNF f para as variáveis $x_1, x_2, ..., x_n$ com cláusulas $C_1, C_2, ..., C_k$, cada uma contendo exatamente três literais distintos, o algoritmo de redução constrói

uma instância $\langle S, t\rangle$ do problema da soma de subconjuntos, tal que f é satisfazível se e somente se existe um subconjunto de S cuja soma seja exatamente t. Sem prejuízo da generalidade, fazemos duas suposições simplificadoras para a fórmula f. A primeira é que nenhuma cláusula contém ambas, uma variável e sua negação, visto que tal cláusula é automaticamente satisfeita por qualquer atribuição de valores para as variáveis. A segunda é que cada variável aparece em, no mínimo, uma cláusula porque não importa qual valor é atribuído a uma variável que não aparece.

A redução cria dois números no conjunto S para cada variável x_i e dois números em S para cada cláusula C_j. Criaremos números em base 10, onde cada número contém $n + k$ dígitos e cada dígito corresponde a uma variável ou a uma cláusula. A base 10 (e outras bases, como veremos) tem a propriedade de que precisamos, de impedir transportes de dígitos mais baixos para dígitos mais altos.

Como mostra a Figura 34.19, construímos o conjunto S e o alvo t da seguinte maneira. Rotulamos cada posição de dígito por uma variável ou por uma cláusula. Os k dígitos menos significativos são rotulados pelas cláusulas e os n dígitos mais significativos são rotulados por variáveis.

- O alvo t tem um 1 em cada dígito rotulado por uma variável, e um 4 em cada dígito rotulado por uma cláusula.
- Para cada variável x_i, o conjunto S contém dois inteiros, v_i e v_i'. Cada v_i e v_i' tem um 1 no dígito rotulado por x_i e 0s nos outros dígitos de variáveis. Se o literal x_i aparece na cláusula C_j, então o dígito rotulado por C_j em v_i contém um 1. Se o literal $\neg x_i$ aparece na cláusula C_j, o dígito rotulado por C_j em v_i' contém um 1. Todos os outros dígitos rotulados por cláusulas em v_i e v_i' são 0.

  Todos os valores v_i e v_i' no conjunto S são únicos. Por quê? Para $l \neq i$, nenhum v_l ou v_l' pode ser igual a v_i e v_i' nos n dígitos mais significativos. Além disso, pelas simplificações que adotamos no início, nenhum v_i e v_i' pode ser igual em todos os k dígitos menos significativos. Se v_i e v_i' fossem iguais, então x_i e $\neg x_i$ teriam de aparecer exatamente no mesmo conjunto de cláusulas. Contudo, combinamos que nenhuma cláusula contém x_i e $\neg x_i$ ao mesmo tempo e que x_i ou $\neg x_i$ aparece em alguma cláusula, e portanto deve haver alguma cláusula C_j para a qual v_i e v_i' são diferentes.
- Para cada cláusula C_j, o conjunto S contém dois inteiros, s_j e s_j'. Cada s_j e s_j' tem 0s em todos os dígitos exceto o dígito identificado por C_j. Para s_j, existe um 1 no dígito C_j e s_j' tem um 2 >nesse dígito. Esses inteiros são "variáveis de folgas", que usamos para conseguir que cada posição de dígito identificada por cláusula alcance o valor alvo de 4.

A simples inspeção da Figura 34.19 demonstra que todos os valores s_j e s_j' em S são únicos no conjunto S.

Observe que a maior soma de dígitos em qualquer posição de dígito é 6, que ocorre nos dígitos identificados por cláusulas (três 1s dos valores v_i e v_i', mais 1 e 2 dos valores s_j e s_j'). Portanto, interpretando esses números em base 10, não pode ocorrer nenhum transporte de dígitos mais baixos para dígitos mais altos.[11]

Podemos executar a redução em tempo polinomial. O conjunto S contém $2n + 2k$ valores, cada um deles com $n + k$ dígitos, e o tempo para produzir cada dígito é polinomial em $n + k$. O alvo t tem $n + k$ dígitos, e a redução produz cada um deles em tempo constante.

Agora mostramos que a fórmula 3-CNF f é satisfazível se e somente se existe um subconjunto $S' \subseteq S$ cuja soma é t. Primeiro, suponha que f tenha uma atribuição que satisfaz. Para $i = 1, 2, ..., n$, se $x_i = 1$ nessa atribuição, então incluímos v_i em S'. Caso contrário, incluímos v_i'. Em outras palavras, incluímos em S' exatamente os valores v_i e v_i' que correspondem a literais com o valor 1 na atribuição. Incluído v_i ou v_i', mas não ambos, para todo i, e como foi inserido 0 nos dígitos rotulados por variáveis em todo s_j e s_j', vemos que para cada dígito rotulado por variável a soma dos valores de S' deve ser 1, o que corresponde aos dígitos do alvo t. Como cada cláusula é satisfeita, a cláusula contém algum literal com o valor 1. Portanto, cada dígito rotulado por uma cláusula tem no mínimo um 1 em sua soma fornecido por um valor v_i ou v_i' em S'. De fato, 1, 2 ou 3 literais podem ter 1 em cada cláusula, e assim cada dígito rotulado por cláusula tem soma de 1, 2 ou 3 dos valores v_i e v_i' em S'. (Por exemplo, na Figura 34.19, os literais $\neg x_1$, $\neg x_2$ e x_3 têm o valor 1 em uma atribuição que satisfaz. Cada uma das cláusulas C_1 e C_4 contém exatamente um desses literais e, assim, juntos, v_1', v_2' e v_3 contribuem com 1 para a soma nos dígitos para C_1 e C_4. A cláusula C_2 contém dois desses literais, e v_1', v_2' e v_3 contribuem com 2 para a soma no dígito correspondente a C_2. A cláusula C_3 contém todos esses três literais, v_1', v_2' e v_3 contribuem com 3 para a soma no dígito correspondente a C_3. Chegamos ao alvo de 4 em cada

dígito identificado pela cláusula C_j incluindo em S' o subconjunto não vazio adequado de variáveis de folga $\{s_j, s_j'\}$. Na Figura 34.19, S' inclui s_1, s_1', s_2', s_3, s_4 e s_4',. Visto que equiparamos o alvo em todos os dígitos da soma e não pode ocorrer nenhum transporte, a soma dos valores de S' é t.

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v_1'	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v_2'	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v_3'	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s_1'	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s_2'	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s_3'	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s_4'	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Figura 34.19 A redução de 3-CNF-SAT a SUBSET-SUM. A fórmula em 3-CNF é $f = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, onde $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ e $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$ e $C_4 = (x_1 \vee x_2 \vee x_3)$. Uma atribuição que satisfaz de f é $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$. O conjunto S produzido pela redução consiste nos números em base 10 mostrados; de cima para baixo, S = {1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2}. O alvo t é 1114444. O subconjunto $S' \subseteq S$ está sombreado em tom mais claro e contém v'_1, v'_2, e v_3, que correspondem à atribuição satisfatória. Também contém variáveis de folgas s_1, s'_1, s'_2, s_3, s_4 e s'_4 para alcançar o valor de alvo 4 nos dígitos identificados por C_1 a C_4.

Agora, suponha que exista um subconjunto $S' \subseteq S$ cuja soma seja t. O subconjunto S' deve incluir exatamente um de v_i e v_i' para cada $i = 1, 2, ..., n$ já que, do contrário, os dígitos identificados por variáveis não somariam 1. Se $v_i \in S'$, definimos $x_i = 1$. Caso contrário, $v_i' \in S'$ e definimos $x_i = 0$. Afirmamos que toda cláusula C_j, para $j = 1, 2, ..., k$, é satisfeita por essa atribuição. Para provar essa afirmação, observe que, para alcançar a soma 4 no dígito identificado por C_j, o subconjunto S' deve incluir no mínimo um valor v_i ou v_i' que tenha 1 no dígito identificado por C_j, já que as contribuições das variáveis de folgas s_j e s_j' juntas somam no máximo 3. Se S' incluir um v_i que tenha um 1 na posição de C_j, então o literal x_i aparece na cláusula C_j. Como definimos $x_i = 1$ quando $v_i \in S'$, a cláusula C_j é satisfeita. Se S' incluir um v_i' que tenha um 1 nessa posição, então o literal $\neg x_i$ aparecerá em C_j. Visto que definimos $x_i = 0$ quando $v_i' \in S'$, a cláusula C_j é novamente satisfeita. Assim, todas as cláusulas de f são satisfeitas, o que conclui a prova.

Exercícios

34.5-1 O ***problema de isomorfismo de subgrafos*** toma dois grafos não dirigidos G_1 e G_2 e pergunta se G_1 é isomorfo a um subgrafo de G_2. Mostre que o problema de isomorfismo de subgrafos é NP-completo.

34.5-2 Dada uma matriz A $m \times n$ de inteiros e um m–vetor inteira, b, o ***problema de programação inteira 0-1*** pergunta se existe um n–vetor de inteiros, x, com elementos no conjunto $\{0, 1\}$ tal que $Ax \leq b$. Prove que a programação inteira 0-1 é NP-completa. (*Sugestão*: Reduza partindo de 3-CNF-SAT.)

34.5-3 O ***problema de programação linear inteira*** é semelhante ao problema de programação inteira 0-1 dado no Exercício 34.5-2, exceto que os valores do vetor x podem ser quaisquer inteiros em vez de somente 0 ou 1. Considerando que o problema de programação inteira 0-1 é NP-difícil, mostre que o problema de programação linear inteira é NP-completo.

34.5-4 Mostre como resolver o problema da soma de subconjuntos em tempo polinomial se o valor alvo t é expresso em unário.

34.5-5 O ***problema de partição de conjuntos*** toma como entrada um conjunto S de números. A questão é se os números podem ser particionados em dois conjuntos A e $A = S - A$ tais que $\sum_{x \in A'} x = \sum_{x \in A'} x$. Mostre que o problema de partição de conjuntos é NP-completo.

34.5-6 Mostre que o problema do caminho hamiltoniano é NP-completo.

34.5-7 O ***problema do ciclo simples de comprimento máximo*** é o problema de determinar um ciclo simples (sem vértices repetidos) de comprimento máximo em um grafo. Formule um problema de decisão relacionado e mostre que esse problema é NP-completo.

34.5-8 No problema de ***satisfazibilidade da meia forma normal 3-conjuntiva*** (***meia 3-CNF***) temos uma fórmula normal 3-conjuntiva f com n variáveis e m cláusulas, onde m é par. Desejamos determinar se existe uma atribuição verdade para as variáveis de f tal que exatamente metade das cláusulas tenha valor 0 e exatamente metade das cláusulas tenha valor 1. Prove que o problema de satisfazibilidade da meia 3-CNF é NP-completo.

Problemas

34-1 ***Conjunto independente***

Um ***conjunto independente*** de um grafo $G = (V, E)$ é um subconjunto $V' \subseteq V$ de vértices, tal que cada aresta em E é incidente em no máximo um vértice em V'. O ***problema do conjunto independente*** é encontrar um conjunto independente de tamanho máximo em G.

a. Formule um problema de decisão relacionado para o problema do conjunto independente e prove que ele é NP-completo. (*Sugestão*: Reduza partindo do problema da clique.)

b. Suponha que você recebeu uma sub-rotina em "caixa-preta" para resolver o problema de decisão que definiu na parte (a). Dê um algoritmo para encontrar um conjunto independente de tamanho máximo. O tempo de execução de seu algoritmo deve ser polinomial em $|V|$ e $|E|$, contando consultas à caixa-preta como uma única etapa. Embora o problema de decisão do conjunto independente seja NP-completo, certos casos especiais são resolvíveis em tempo polinomial.

c. Dê um algoritmo eficiente para resolver o problema do conjunto independente quando cada vértice em G tem grau 2. Analise o tempo de execução e prove que seu algoritmo funciona corretamente.

d. Dê um algoritmo eficiente para resolver o problema do conjunto independente quando G é bipartido. Analise o tempo de execução e prove que seu algoritmo funciona corretamente. (*Sugestão*: Use os resultados da Seção 26.3.)

34-2 Bonnie e Clyde

Bonnie e Clyde acabaram de assaltar um banco. Eles têm uma sacola de dinheiro e querem reparti-lo. Para cada um dos cenários a seguir, dê um algoritmo de tempo polinomial ou prove que o problema é NP-completo. A entrada em cada caso é uma lista dos n itens na sacola, junto com o valor de cada um.

a. A sacola contém n moedas, mas somente duas denominações diferentes: algumas moedas valem x dólares e algumas valem y dólares. Bonnie e Clyde desejam dividir o dinheiro em partes exatamente iguais.

b. A sacola contém n moedas, com um número arbitrário de denominações diferentes, mas cada denominação é uma potência inteira não negativa de 2, isto é, os valores possíveis das denominações são 1 dólar, 2 dólares, 4 dólares etc. Bonnie e Clyde desejam dividir o dinheiro em partes exatamente iguais.

c. A sacola contém n cheques, que, por uma coincidência incrível, são nominais a "Bonnie ou Clyde". Eles desejam dividir os cheques de modo que cada um receba exatamente a mesma quantia.

d. A sacola contém n cheques como na parte (c), mas dessa vez Bonnie e Clyde estão dispostos a aceitar uma divisão em que a diferença não seja maior que 100 dólares.

34-3 Coloração de grafos

Fabricantes de mapas tentam usar o mínimo de cores possível para colorir países em um mapa, desde que dois países que compartilhem uma fronteira não tenham a mesma cor.

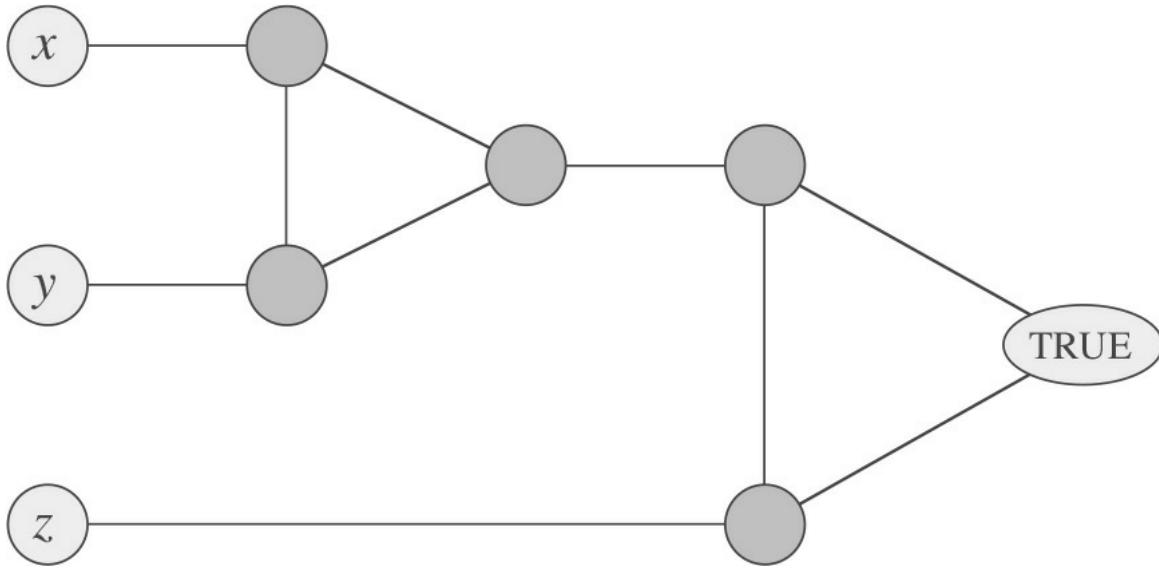
Podemos modelar esse problema com um grafo não dirigido G = (V, E) no qual cada vértice representa um país e os vértices cujos respectivos países compartilham uma fronteira são adjacentes. Então ***colorir um grafo com k cores*** é uma função $c : V \rightarrow \{1, 2, \ldots, k\}$ tal que $c(u) \neq c(v)$ para toda aresta $(u, v) \in E$. Em outras palavras, os números 1, 2, ..., k representam as k cores, e vértices adjacentes devem ter cores diferentes. O ***problema da coloração de grafos*** é determinar o número mínimo de cores necessárias para colorir um dado grafo.

a. Dê um algoritmo eficiente para colorir um grafo com duas cores, se ele existir.

b. Expresse o problema de colorir grafos como um problema de decisão. Mostre que seu problema de decisão é resolvível em tempo polinomial se e somente se o problema da coloração de grafos é resolvível em tempo polinomial.

c. Seja 3-COLOR a linguagem que é o conjunto de grafos que podem ser coloridos em três cores). Mostre que, se 3-COLOR é NP-completo, seu problema de decisão da parte (b) é NP-completo.

Para provar que 3-COLOR é NP-completo, usamos uma redução de 3-CNF-SAT. Dada uma fórmula f, de m cláusulas para n variáveis $x_1, x_2, ..., x_n$, construímos um grafo $G = (V, E)$ da seguinte maneira. O conjunto V consiste em um vértice para cada variável, um vértice para a negação de cada variável, cinco vértices para cada cláusula e três vértices especiais: TRUE, FALSE e RED. As arestas do grafo são de dois tipos: arestas "literais", que são independentes das cláusulas, e arestas "cláusulas", que dependem das cláusulas. As arestas literais formam um triângulo nos vértices especiais e também formam um triângulo em x_i, $\neg x_i$, e RED para $i = 1, 2, ..., n$.

d. Demonstre que, em qualquer problema c de colorir em três cores um grafo que contém arestas literais, exatamente uma de uma variável e sua negação é colorida c(TRUE) e a outra é colorida c(FALSE). Demonstre que, para qualquer atribuição verdade de f, existe uma coloração em três cores do grafo que contém apenas as arestas literais.

O widget mostrado na Figura 34.20 ajuda a impor a condição correspondente a uma cláusula ($x \vee y \vee z$). Cada cláusula requer uma cópia única dos cinco vértices que estão sombreados em tom mais escuro na figura; eles se ligam aos literais da cláusula e ao vértice especial TRUE da maneira mostrada.

e. Demonstre que, se cada x, y e z é colorido c(TRUE) ou c(FALSE), o widget pode pode ser colorido em três cores se e somente se no mínimo um de x, y ou z é colorido c(TRUE).

f. Conclua a prova de que 3-COLOR é NP-completo.

34-4 Escalonamento com lucros e prazos finais

Suponha que tenhamos uma máquina e um conjunto de n tarefas $a_1, a_2, ..., a_n$, e cada uma requeira tempo na máquina. Cada tarefa a_j requer t_j unidades de tempo na máquina (seu tempo de processamento), rende um lucro p_j e tem um prazo final d_j. A máquina pode processar somente uma tarefa por vez, e a tarefa a_j deve ser executada ininterruptamente por t_j unidades de tempo consecutivas. Se concluirmos a tarefa a_j em seu prazo final d_j, receberemos um lucro p_j, mas se a concluirmos após seu prazo final não obteremos nenhum lucro. Por ser um problema de otimização, temos os tempos de processamento, lucros e prazos finais para um conjunto de n tarefas e desejamos determinar um escalonamento que conclua todas as tarefas e retorne o maior lucro possível. Os tempos de processamento, lucros e prazos finais são números negativos.

a. Enuncie este problema como um problema de decisão.

b. Mostre que o problema de decisão é NP-completo.

c. Dê um algoritmo de tempo polinomial para o problema de decisão supondo que todos os tempos de processamento são inteiros de 1 a n. (*Sugestão:* Use programação dinâmica.)

d. Dê um algoritmo de tempo polinomial para o problema de otimização, supondo que todos os tempos de processamento são inteiros de 1 a n.

Figura 34.20 O *widget* correspondente a uma cláusula ($x \lor y \lor z$), usado no Problema 34-3.

NOTAS DO CAPÍTULO

O livro de Garey e Johnson [129] é um maravilhoso guia para a NP-completude por discutir a teoria a fundo e fornecer um catálogo de muitos problemas que eram conhecidos como NP-completos em 1979. A prova do Teorema 34.13 foi adaptada desse livro, e a lista de domínios de problemas NP-completos no início da Seção 34.5 foi extraída de seu sumário. Johnson escreveu uma série de 23 colunas no *Journal of Algorithms* entre 1981 e 1992 relatando novos desenvolvimentos no estudo da NP-completude. Hopcroft, Motwani e Ullman [177], Lewis e Papadimitriou [236], Papadimitriou [270] e Sipser [317] trazem bons tratamentos da NP-completude no contexto da teoria da complexidade. A NP-completude e várias reduções também aparecem em livros por Aho, Hopcroft e Ullman [5], Dasgupta, Papadimitriou e Varizani [82], Johnsonbaugh e Schaefer [193] e Kleinberg e Tardos [208].

A classe P foi introduzida em 1964 por Cobham [72] e, independentemente, em 1965 por Edmonds [100], que também apresentou a classe NP e conjeturou que P $\neq$ NP. A noção da NP-completude foi proposta em 1971 por Cook [75], que deu as primeiras provas da NP-completude para satisfazibilidade de fórmulas e satisfazibilidade 3-CNF. Levin [234] descobriu independentemente a noção, dando uma prova da NP-completude para um problema de ladrilhamento. Karp [199] introduziu a metodologia de reduções em 1972 e demonstrou a rica variedade de problemas NP-completos. O artigo de Karp incluía as provas originais da NP-completude dos problemas da clique, da cobertura de vértices e do ciclo hamiltoniano. Desde então, muitos pesquisadores provaram que milhares de problemas eram NP-completos. Em uma conversa durante uma reunião para comemorar os 60 anos de Karp em 1995, Papadimitriou observou que "aproximadamente 6.000 artigos por ano são publicados com a expressão 'NP-completo' em seu título, sumário ou lista de palavras-chave. Isso é mais do que cada um dos termos 'compilador', 'banco de dados', 'especialista', 'rede neural' ou 'sistema operacional'".

O trabalho recente em teoria da complexidade lançou alguma luz sobre a complexidade do cálculo de soluções aproximadas. Ele apresentou uma nova definição de NP, usando "provas probabilisticamente verificáveis". Essa nova definição implica que, para problemas como o do clique, da cobertura de vértices, do caixeiro viajante com a desigualdade triangular e muitos outros, o cálculo de boas soluções aproximadas é NP-difícil e, consequentemente, não é mais fácil que o cálculo de soluções ótimas. Uma introdução a esse assunto pode ser encontrada na tese de Arora [20], em um capítulo de Arora e Lund em Hochbaum [172], um levantamento por Arora [21], um livro editado por Mayr, Prömel e Steger [246] e em um artigo de pesquisa de Johnson [191].

[1] Consulte em Hopcrotf e Ullman [180] ou Lewis e Papadimitriou [236] um tratamento completo do modelo da máquina de Turing.

[2] O contradomínio de e não precisa ser cadeias *binárias*; qualquer conjunto de cadeias sobre um alfabeto finito que tenha pelo menos dois símbolos servirá.

[3] Supomos que a saída do algoritmo está separada de sua entrada. Como demora no mínimo uma etapa de tempo para produzir cada bit da saída e o algoritmo leva $O(T(n))$ etapas de tempo, o tamanho da saída é $O(T(n))$.

[4] Denotamos por $\{0, 1\}^*$ o conjunto de todas as cadeias compostas de símbolos do conjunto $\{0, 1\}$.

[5] Tecnicamente, também exigimos que as funções f_{12} e f_{21} "mapeiem não instâncias para não instâncias". Uma ***não instância*** de uma codificação e é uma cadeia $x \in \{0, 1\}^*$ tal que não existe nenhuma instância i para a qual $e(i) = x$. É preciso que $f_{12}(x) = y$ para toda não instância x da codificação e_1, onde y é alguma não instância de e_2, e que $f_{21}(x') = y'$ para toda não instância x' de e_2, onde y' é alguma não instância de e_1.

[6] Para saber mais sobre classes de complexidade, consulte o artigo seminal de Hartmanis e Stearns [162].

[7] Em uma carta a seu amigo John T. Graves, datada de 17 de outubro de 1856, Hamilton [157, p. 624] escreveu: "Descobri que alguns jovens se divertem muito com um novo jogo matemático que o Icosion fornece, no qual uma pessoa fixa cinco alfinetes em quaisquer cinco pontos consecutivos (...) e o outro jogador tenta inserir, o que pela teoria descrita nesta carta sempre pode ser feito, quinze alfinetes em sucessão cíclica, de modo a passar por todos os pontos e terminar na proximidade imediata do alfinete com o qual o antagonista começou.

[8] O nome "NP" significa, em inglês, "tempo polinomial não determinístico" (*nondeterministic polynomial time*). A classe NP foi estudada originalmente no contexto do não determinismo, mas este livro usa a noção um pouco mais simples, ainda que equivalente, de verificação. Hopcroft e Ullman [180] dão uma boa apresentação da NP-completude em termos de modelos não determinísticos de computação.

[9] Por outro lado, se o tamanho do circuito C é $\Theta(2^k)$, então um algoritmo cujo tempo de execução é $O(2^k)$ tem um tempo de execução que é polinomial em relação ao tamanho do circuito. Mesmo que P ≠ NP, essa situação não contradiria a NP-completude do problema; a existência de um algoritmo de tempo polinomial para um caso especial não implica que existe um algoritmo de tempo polinomial para todos os casos.

[10] Tecnicamente, definimos um ciclo em termos de vértices em vez de arestas (veja a Seção B.4). Por questão de clareza, abusamos da notação aqui e definimos o ciclo hamiltoniano em termos de arestas.

[11] De fato, qualquer base b, onde $b \geq 7$, serviria. A instância no início desta subseção é o conjunto S e alvo t na Figura 34.19 interpretados em base 7, sendo S listado em sequência ordenada.

35 ALGORITMOS DE APROXIMAÇÃO

Muitos problemas de significado prático são NP-completos, mas apesar disso são demasiadamente importantes para que os abandonemos simplesmente porque não sabemos como determinar uma solução ótima em tempo polinomial. Mesmo quando um problema é NP-completo, ainda pode haver esperança. Temos no mínimo três modos de contornar a NP-completude. O primeiro é que, se as entradas reais são pequenas, um algoritmo com tempo de execução exponencial pode ser perfeitamente satisfatório. O segundo é que podemos conseguir isolar casos especiais importantes que podemos resolver em tempo polinomial. O terceiro é que poderíamos encontrar abordagens para determinar soluções *quase ótimas* em tempo polinomial (seja no pior caso ou no caso esperado). Na prática, quase ótimo muitas vezes é suficientemente bom. Denominamos um algoritmo que retorna soluções quase ótimas ***algoritmo de aproximação***. Este capítulo apresenta algoritmos de aproximação de tempo polinomial para vários problemas NP-completos.

Razões de desempenho para algoritmos de aproximação

Suponha que estejamos trabalhando em um problema de otimização em que cada solução potencial tenha um custo positivo e que desejamos encontrar uma solução quase ótima. Dependendo do problema, podemos definir uma solução ótima como uma solução com o máximo custo possível ou uma solução com o mínimo custo possível; isto é, o problema pode ser um problema de maximização ou um problema de minimização.

Dizemos que um algoritmo para um problema tem uma ***razão de aproximação*** $r(n)$ se, para qualquer entrada de tamanho n, o custo C da solução produzida pelo algoritmo está a menos de um fator $r(n)$ do custo C^* de uma solução ótima:

$$máx\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n) \tag{35.1}$$

Se um algoritmo consegue uma razão de aproximação de $r(n)$, nós o denominamos ***algoritmo de r(n)-aproximação.*** As definições de razão de aproximação e algoritmo de aproximação $r(n)$ se aplicam a problemas de minimização, bem como de maximização. Para um problema de maximização, $0 < C \leq C^*$, e a razão C^*/C dá o fator que indica quanto o custo de uma solução ótima é maior que o custo da solução aproximada. De modo semelhante, para um problema de minimização, $0 < C^* \leq C$, e a razão C/C^* dá o fator que indica quanto o custo da solução aproximada é maior que o custo de uma solução ótima. Como supomos que todas as soluções têm custo positivo, essas razões são sempre bem definidas. A razão de aproximação de um algoritmo de aproximação nunca é menor que 1, já que $C/C^* < 1$ implica $C^*/C > 1$. Então, um algoritmo de aproximação[1] 1 produz uma solução ótima, e um algoritmo de aproximação com uma razão de aproximação grande pode retornar uma solução muito pior que a solução ótima.

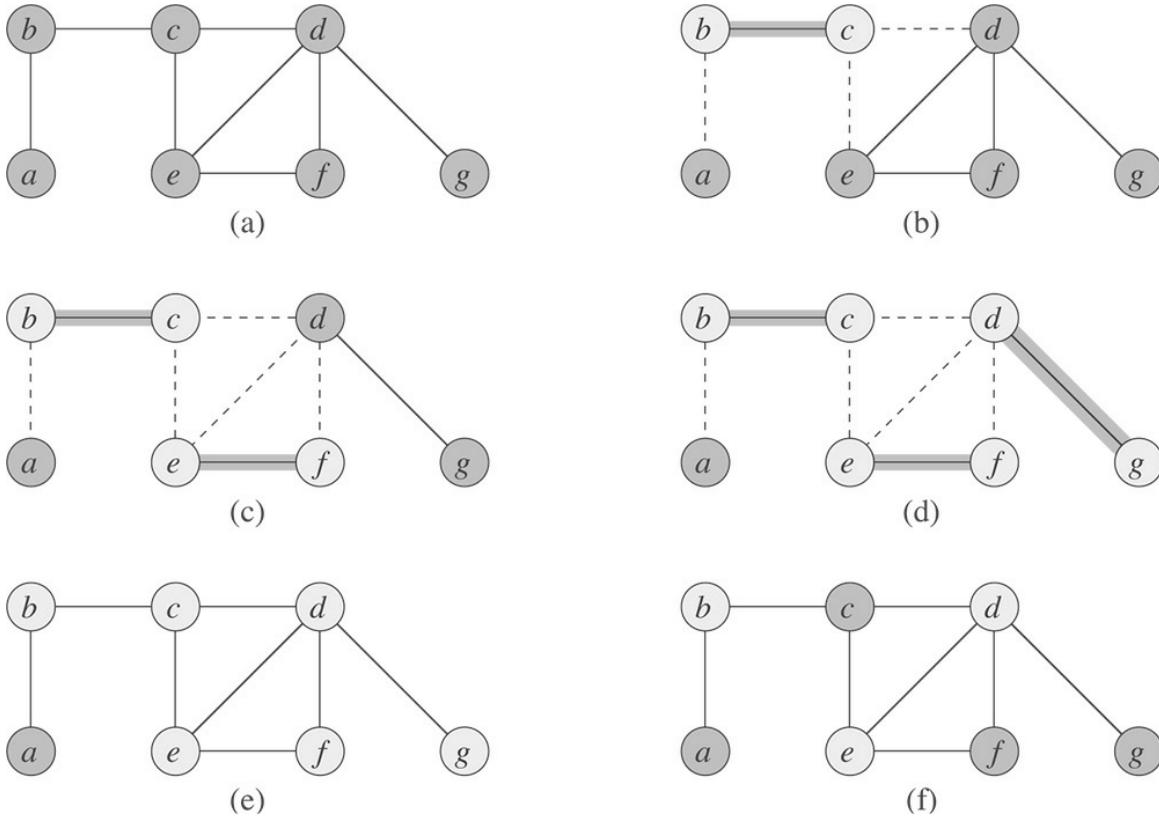
Para muitos problemas temos algoritmos de aproximação de tempo polinomial com razões de aproximação pequenas e constantes, embora para outros problemas os algoritmos de aproximação de tempo polinomial mais conhecidos tenham razões de aproximação que crescem em função do tamanho da entrada n. Um exemplo de tal problema é o problema da cobertura de conjuntos apresentado na Seção 35.3.

Alguns problemas NP-completos permitem algoritmos de aproximação de tempo polinomial que podem conseguir razões de aproximação cada vez melhores, usando cada vez mais tempo de computação. Isto é, podemos permutar tempo de computação e qualidade da aproximação. Um exemplo é o problema da soma de subconjuntos estudado na Seção 35.5. Essa situação é bastante importante para merecer um nome exclusivo.

Um ***esquema de aproximação*** para um problema de otimização é um algoritmo de aproximação que adota como entrada não somente uma instância do problema, mas também um valor $e > 0$ tal que, para qualquer e fixo, o esquema é um algoritmo de $(1 + e)$-aproximação. Dizemos que um esquema de aproximação é um ***esquema de aproximação de tempo polinomial*** se, para qualquer $e > 0$ fixo, o esquema é executado em tempo polinomial no tamanho n de sua instância de entrada.

O tempo de execução de um esquema de aproximação de tempo polinomial pode aumentar muito rapidamente à medida que e diminui. Por exemplo, o tempo de execução de um esquema de aproximação de tempo polinomial poderia ser $O(n_{2/e})$. No caso ideal, se e diminui por um fator constante, o tempo de execução para obter a aproximação desejada não deve aumentar mais do que um fator constante (embora não necessariamente o mesmo fator constante pelo qual e diminuiu).

Dizemos que um esquema de aproximação é um ***esquema de aproximação de tempo completamente polinomial*** se é um esquema de aproximação e seu tempo de execução é polinomial em $1/e$, bem como no tamanho n da instância de entrada. Por exemplo, o esquema poderia ter um tempo de execução $O((1/e)^2 n_3)$. Com tal esquema, qualquer redução de fator constante em e pode vir acompanhada por um aumento correspondente de fator constante no tempo de execução.

Resumo do capítulo

As quatro primeiras seções deste capítulo apresentam alguns exemplos de algoritmos de aproximação de tempo polinomial para problemas NP-completos, e a quinta seção apresenta um esquema de aproximação de tempo completamente polinomial. A Seção 35.1 começa com um estudo do problema de cobertura de vértices, um problema de minimização NP-completo que tem um algoritmo de aproximação com uma razão de aproximação igual a 2. A Seção 35.2 apresenta um algoritmo de aproximação com razão 2 para o caso do problema do caixeiro-viajante, no qual a função custo satisfaz a desigualdade triangular. Mostra também que, sem a desigualdade triangular, para qualquer constante $r \geq 1$, um algoritmo de r-aproximação não pode existir a menos que P = NP. Na Seção 35.3, mostramos como usar um método guloso como um algoritmo de aproximação efetivo para o problema da cobertura de conjuntos, obtendo uma cobertura cujo custo é na pior das hipóteses um fator logarítmico maior que o custo ótimo. A Seção 35.4 apresenta mais dois algoritmos de aproximação. Primeiro, estudamos a versão de otimização da satisfazibilidade 3-CNF e damos um algoritmo aleatorizado simples que produz uma solução com uma razão de aproximação esperada de 8/7. Depois examinamos uma variante ponderada do problema de cobertura de vértices e mostramos como usar programação linear para desenvolver um algoritmo de aproximação 2. Finalmente, a Seção 35.5 apresenta um esquema de aproximação de tempo completamente polinomial para o problema da soma de subconjuntos.

35.1 O PROBLEMA DE COBERTURA DE VÉRTICES

A Seção 34.5.2 definiu o problema de cobertura de vértices e provou que ele é NP-completo. Lembre-se de que uma ***cobertura de vértices*** de um grafo não dirigido $G = (V, E)$ é um subconjunto $V' \subseteq V$ tal que, se (u, v) é uma aresta de G, então $u \in V'$ ou $v \in V'$ (ou ambos). O tamanho de uma cobertura de vértices é o número de vértices que ela contém.

O ***problema de cobertura de vértices*** é encontrar uma cobertura de vértices de tamanho mínimo em um grafo não dirigido dado. Denominamos tal cobertura de vértices ***cobertura de vér- tices ótima***. Esse problema é a versão de otimização de um problema de decisão NP-completo.

Embora não saibamos como determinar uma cobertura de vértices ótima em um grafo G em tempo polinomial, podemos encontrar eficientemente uma cobertura de vértices que é quase ótima. O algoritmo de aproximação a seguir adota como entrada um grafo não dirigido G e retorna uma cobertura de vértices para a qual é garantido que seu tamanho não é mais de duas vezes maior do que a cobertura de vértices ótima.

Figura 35.1 A operação de Approx-Vertex-Cover. **(a)** O grafo de entrada G, que tem sete vértices e oito arestas. **(b)** A aresta (b, c), mostrada em sombreado de tom mais escuro, é a primeira aresta escolhida por Approx-Vertex-Cover. Os vértices b e c, mostrados em sombreado de tom mais claro, são adicionados ao conjunto C que contém a cobertura de vértices que está sendo criada. As arestas (a, b), (c, e) e (c, d), mostradas em linhas tracejadas, são eliminadas visto que agora estão cobertas por algum vértice em C. **(c)** A aresta (e, f) é escolhida; os vértices e e f são adicionados a C. **(d)** A aresta (d, g) é escolhida; os vértices d e g são adicionados a C. **(e)** O conjunto C, que é a cobertura de vértices produzida por Approx-Vertex-Cover, contém os seis vértices b, c, d, e, f, g. **(f)** A cobertura de vértices ótima para esse problema contém apenas três vértices: b, d e e.

A Figura 35.1 ilustra a operação de Approx-Vertex-Cover em um grafo de exemplo. A variável C contém a cobertura de vértices que está sendo construída. A linha 1 inicializa C como o conjunto vazio. A linha 2 define E' como uma cópia do conjunto de arestas $E[G]$ do grafo. O laço nas linhas 3–6 escolhe repetidamente uma aresta (u, v) de E', adiciona suas extremidades u e v a C e elimina todas as arestas em E' que são cobertas por u ou v. O tempo de execução desse algoritmo é $O(V + E)$, usando listas de adjacências para representar E'.

Teorema 35.1

Approx-Vertex-Cover é um algoritmo polinomial de 2-aproximação.

Prova Já mostramos que Approx-Vertex-Cover é executado em tempo polinomial.

O conjunto C de vértices que é retornado por Approx-Vertex-Cover é uma cobertura de vértices, visto que o algoritmo roda até que toda aresta em $E[G]$ tenha sido coberta por algum vértice em C.

Para ver que Approx-Vertex-Cover devolve uma cobertura de vértices que é no máximo duas vezes o tamanho de uma cobertura ótima, seja A o conjunto de arestas que foram escolhidas na linha 4 de Approx-Vertex-Cover. Para cobrir as arestas em A, qualquer cobertura de vértices — em particular uma cobertura ótima C^* — deve incluir no mínimo uma extremidade de cada aresta em A. Duas arestas em A não compartilham uma extremidade, já que, uma vez escolhida uma aresta na linha 4, todas as outras arestas que são incidentes em suas extremidades são eliminadas de E' na linha 6. Assim, não há duas arestas em A cobertas pelo mesmo vértice de C_*, e temos o limite inferior

$$|C^*| \geq |A| \tag{35.2}$$

para o tamanho de uma cobertura de vértices ótima. Cada execução da linha 4 escolhe uma aresta para a qual nenhuma de suas extremidades já está em C, o que produz um limite superior (na verdade, um limite superior exato) para o tamanho da cobertura de vértices devolvida:

$$|C| = 2\ |A|. \tag{35.3}$$

Combinando as equações (35.2) e (35.3), obtemos

$$\begin{aligned} |C| &= 2\ |A| \\ &\leq 2|C^*|\ , \end{aligned}$$

provando assim o teorema.

Vamos refletir sobre essa prova. A princípio, poderíamos imaginar como é possível provar que o tamanho da cobertura de vértices devolvida por Approx-Vertex-Cover é no máximo duas vezes o tamanho de uma cobertura de vértices ótima, já que nem mesmo sabemos qual é o tamanho da cobertura de vértices ótima. Em vez de querermos saber qual é o tamanho exato de uma cobertura de vértices ótima, recorremos a um limite inferior para o tamanho. Como o Exercício 35.1-2 lhe pede para mostrar, o conjunto A de arestas que a linha 4 de Approx-Vertex-Cover seleciona é realmente um emparelhamento maximal no grafo G. (Um ***emparelhamento maximal*** é um emparelhamento que não é um subconjunto próprio de qualquer outro emparelhamento.) O tamanho de um emparelhamento maximal é, como demonstramos na prova do Teorema 35.1, um limite inferior para o tamanho de uma cobertura de vértices ótima. O algoritmo retorna uma cobertura de vértices cujo tamanho é no máximo duas vezes o tamanho do emparelhamento maximal A. Determinando a razão entre o tamanho da solução retornada e o limite inferior, obtemos nossa razão de aproximação. Utilizaremos essa metodologia também em seções posteriores.

Exercícios

35.1-1 Dê um exemplo de grafo para o qual Approx-Vertex-Cover sempre produz um solução subótima.

35.1-2 Prove que o conjunto de arestas escolhido na linha 4 de Approx-Vertex-Cover forma um emparelhamento maximal no grafo G.

35.1-3 ★

O professor Bündchen propõe a seguinte heurística para resolver o problema de cobertura de vértices. Selecione repetidamente um vértice de grau mais alto e elimine todas as suas arestas incidentes. Dê um exemplo para mostrar que a heurística do professor não tem uma razão de aproximação de 2. (*Sugestão:* Experimente um grafo bipartido com vértices de grau uniforme à esquerda e vértices de grau variado à direita.)

35.1-4 Dê um algoritmo guloso eficiente para determinar uma cobertura de vértices ótima para uma árvore em tempo linear.

35.1-5 Pela prova do Teorema 34.12, sabemos que o problema de cobertura de vértices e o problema NP-completo do clique são complementares no sentido de que uma cobertura de vértices ótima é o complemento de um clique de tamanho máximo no grafo complemento. Essa relação implica que existe um algoritmo de aproximação de tempo polinomial com razão de aproximação constante para o problema do clique? Justifique sua resposta.

35.2 O PROBLEMA DO CAIXEIRO-VIAJANTE

No problema do caixeiro-viajante apresentado na Seção 34.5.4, temos um grafo não dirigido completo $G = (V, E)$ que tem um custo inteiro não negativo $c(u, v)$ associado a cada aresta $(u, v) \in E$ e devemos encontrar um ciclo hamiltoniano de G com custo mínimo. Como uma extensão de nossa notação, seja $c(A)$ o custo total das arestas no subconjunto $A \subseteq E$:

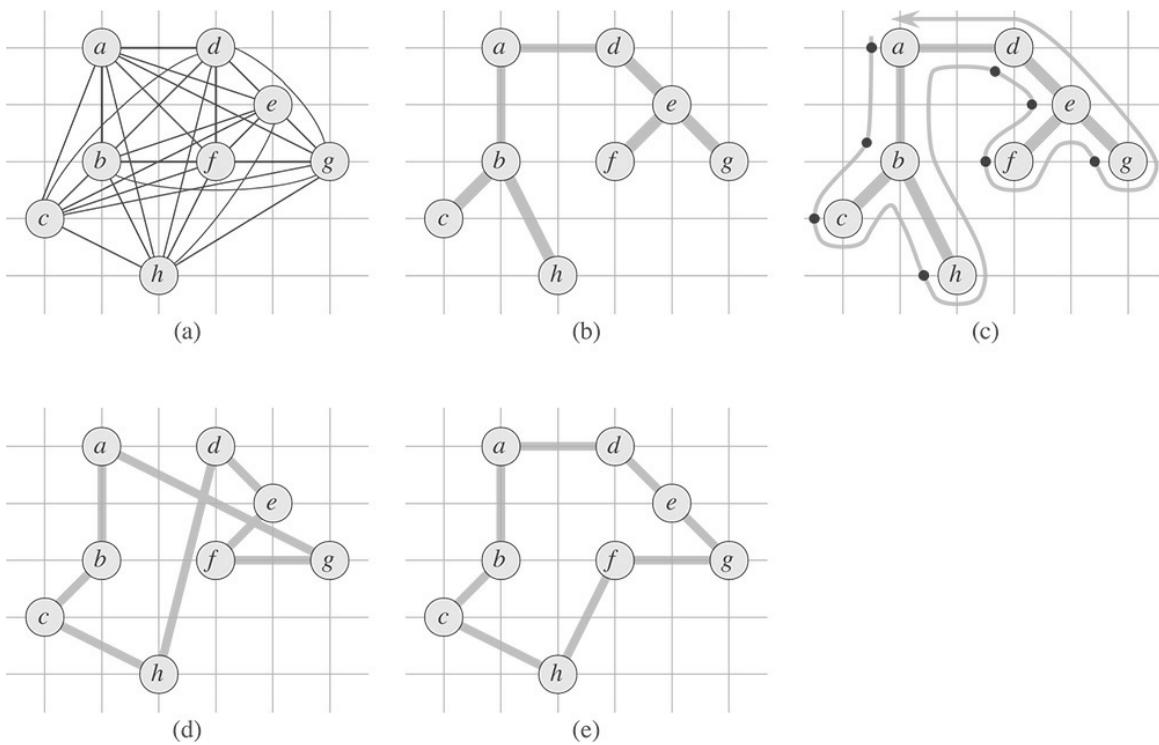
$$c(A) = \sum_{(u,v)\in A} c(u,v) .$$

Em muitas situações práticas, o modo menos caro de ir de um lugar u a um lugar w é ir diretamente, sem nenhuma etapa intermediária. Em outros termos, eliminar uma parada intermediária nunca aumenta o custo. Formalizamos essa noção dizendo que a função custo c satisfaz a ***desigualdade triangular*** se, para todos os vértices $u, v, w \in V$,

$$c(u, w) \leq c(u, v) + c(v, w) .$$

A desigualdade triangular parece naturalmente válida, e ela é automaticamente satisfeita em muitas aplicações. Por exemplo, se os vértices do grafo são pontos no plano e o custo de viajar entre dois vértices é a distância euclidiana comum entre eles, a desigualdade triangular é satisfeita. Além disso, muitas funções custo além da distância euclidiana satisfazem a desigualdade triangular.)

Como mostra o Exercício 35.2-2, o problema do caixeiro-viajante é NP-completo mesmo se exigirmos que a função custo satisfaça a desigualdade triangular. Assim, não devemos esperar encontrar um algoritmo de tempo polinomial para resolver esse problema com exatidão. Em vez disso, procuramos bons algoritmos de aproximação.

Na Seção 35.2.1 examinamos um algoritmo de aproximação 2 para o problema do caixeiro-viajante com a desigualdade triangular. Na Seção 35.2.2 mostraremos que, sem a desigualdade triangular, não existe um algoritmo de aproximação de tempo polinomial com uma razão de aproximação constante a menos que P = NP.

35.2.1 O PROBLEMA DO CAIXEIRO-VIAJANTE COM A DESIGUALDADE TRIANGULAR

Aplicando a metodologia da seção anterior, calcularemos primeiro uma estrutura — uma árvore geradora mínima — cujo peso dá um limite inferior para o comprimento de um passeio ótimo do caixeiro-viajante. Depois usaremos a árvore geradora mínima para criar um passeio cujo custo não é mais que duas vezes o peso da árvore geradora mínima, desde que a função custo satisfaça a desigualdade triangular. O algoritmo a seguir implementa essa abordagem, chamando o algoritmo de árvore geradora mínima MST-Prim da Seção 23.2 como uma sub-rotina.

```
APPROX-TSP-TOUR(G, c)
1  selecione um vértice r ∈ G.V para ser um vértice "raiz"
2  calcule uma árvore geradora mínima T para G partindo da raiz r
      usando MST-PRIM(G, c, r)
3  seja H a lista de vértices, ordenados de acordo com a ordem em que foram visitados
      pela primeira vez em um passeio de pré-ordem em T
4  return o ciclo hamiltoniano H
```

Lembre-se de que, na Seção 12.1, dissemos que um passeio de árvore em pré-ordem visita recursivamente todos os vértices da árvore e insere um vértice na lista de vértices quando o encontra pela primeira vez, antes de visitar qualquer de seus filhos.

Figura 35.2 A operação de Approx-tSp-tour. **(a)** Um grafo não direcionado completo. Os vértices estão em interseções de linhas de grade inteiras. Por exemplo, f está uma unidade à direita e duas unidades acima de h. A função custo entre dois pontos é a distância euclidiana normal. **(b)** Uma árvore geradora mínima T do grafo completo, como calculada por MSt-prIM. O vértice a é o vértice raiz. São mostradas apenas arestas que estão na árvore geradora mínima. Acontece que os vértices são rotulados de tal modo que são adicionados à árvore principal por MSt-prIM em ordem alfabética. **(c)** Um passeio de T, começando em a. Um passeio completo da árvore visita os vértices na ordem a, b, c, b, h, b, a, d, e, f, e, g, e, d, a. Um passeio em pré-ordem de T acrescenta um vértice à lista de vértices apenas quando ele é encontrado pela primeira vez, conforme indicado pelo ponto ao lado de cada vértice, produzindo a ordenação a, b, c, h, d, e, f, g. **(d)** Um passeio obtido pela visita aos vértices na ordem dada pelo passeio em pré-ordem, que é o passeio H retornado por Approx-tSp-tour. Seu custo total é aproximadamente 19,074. **(e)** Um passeio ótimo H* para o grafo completo dado. Seu custo total é aproximadamente 14,715.

A Figura 35.2 ilustra a operação de Approx-TSP-Tour. A parte (a) da figura mostra um grafo não dirigido completo e a parte (b) mostra a árvore geradora mínima T que MST-Prim fez crescer partindo do vértice de raiz a. A parte (c) mostra como um passeio em pré-ordem de T visita os vértices, e a parte (d) apresenta o passeio correspondente, que é o passeio devolvido por Approx-TSP-Tour. A parte (e) exibe um passeio ótimo, que é aproximadamente 23% mais curto.

Pelo Exercício 23.2-2, até mesmo com uma implementação simples de MST-Prim, o tempo de execução de Approx-TSP-Tour é Q(V 2). Agora, mostramos que, se a função custo para uma instância do problema do caixeiro-viajante satisfaz a desigualdade triangular, então Approx-TSP-Tour retorna um passeio cujo custo não é mais do que duas vezes o custo de um passeio ótimo.

Teorema 35.2

Approx-TSP-Tour é um algoritmo polinomial de 2-aproximação para o problema do caixeiro-viajante com a desigualdade de triângulos.

Prova Já vimos que Approx-TSP-Tour é executado em tempo polinomial.

Seja H^* um passeio ótimo para o conjunto de vértices dado. Obtemos uma árvore geradora eliminando qualquer aresta de um passeio, e o custo de cada aresta é não negativo. Portanto, o peso da árvore geradora mínima T calculado na linha 2 de Appox-TSP-Tour dá um limite inferior para o custo de um passeio ótimo, ou seja,

$$c(T) \leq c(H^*) . \tag{35.4}$$

Um ***passeio completo*** de T insere vértices na lista de vértices quando eles são visitados pela primeira vez e também sempre que voltamos a eles após uma visita a uma subárvore. Vamos denominar esse passeio completo W. O passeio completo de nosso exemplo dá a ordem

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a .$$

Visto que o passeio completo percorre todas as arestas de T exatamente duas vezes, temos (estendendo nossa definição do custo c da maneira natural para tratar conjuntos múltiplos de arestas)

$$c(W) = 2c(T) . \tag{35.5}$$

As equações (35.4) e (35.5) implicam que

$$c(W) \leq 2c(H^*) , \tag{35.6}$$

e portanto, o custo de W está a menos de um fator 2 do custo de um passeio ótimo.

Infelizmente, o passeio completo W em geral não é um ciclo simples, já que visita alguns vértices mais de uma vez. Contudo, pela desigualdade triangular, podemos eliminar uma visita a qualquer vértice partindo de W, e o custo não aumenta. (Se eliminarmos um vértice v de W entre visitas a u e w, a ordenação resultante especifica ir diretamente de u para w.) Aplicando essa operação repetidas vezes, podemos eliminar tudo de W, exceto a primeira visita a cada vértice. Em nosso exemplo, isso deixa a ordenação

$$a, b, c, h, d, e, f, g .$$

Essa ordenação é a mesma que foi obtida por um passeio em pré-ordem da árvore T. Seja H o ciclo correspondente a esse passeio em pré-ordem. Ele é um ciclo hamiltoniano, visto que todo vértice é visitado exatamente uma vez, e é de fato o ciclo calculado por Approx-TSP-Tour. Visto que H é obtido pela eliminação de vértices do passeio completo W, temos

$$c(H) \leq c(W) . \tag{35.7}$$

Combinando as desigualdades (35.6) e (35.7) resultam $c(H) \leq 2c(H_*)$, o que conclui a prova.

Apesar da bela razão de aproximação dada pelo Teorema 35.2, normalmente Approx-TSP-Tour não é a melhor escolha prática para esse problema. Existem outros algoritmos de aproximação que, em geral, funcionam muito melhor na prática (consulte as referências no final deste capítulo).

35.2.2 O PROBLEMA GERAL DO CAIXEIRO-VIAJANTE

Se eliminarmos a hipótese de que a função custo c satisfaz a desigualdade triangular, então não podemos encontrar bons passeios aproximados em tempo polinomial, a menos que P = NP.

Teorema 35.3

Se P $\neq$ NP, então para toda constante $r \geq 1$, não existe nenhum algoritmo de aproximação de tempo polinomial com razão de aproximação r para o problema geral do caixeiro-viajante.

Prova A prova é por contradição. Suponha ao contrário que, para algum número $r \geq 1$ exista um algoritmo de aproximação de tempo polinomial A com razão de aproximação r. Sem perda da generalidade, supomos que r é um inteiro, arredondando-o se necessário. Então, mostraremos como usar A para resolver instâncias do problema do ciclo hamiltoniano (definido na Seção 34.2) em tempo polinomial. Visto que o Teorema 34.13 nos diz que o problema do ciclo hamiltoniano é NP-completo, o Teorema 34.4 implica que, se pudermos resolvê-lo em tempo polinomial, P = NP.

Seja $G = (V, E)$ uma instância do problema do ciclo hamiltoniano. Desejamos determinar eficientemente se G contém um ciclo hamiltoniano fazendo uso do algoritmo de aproximação hipotético A. Transformamos G em uma instância do problema do caixeiro-viajante da maneira ilustrada a seguir. Seja $G' = (V, E')$ o grafo completo em V, isto é,

$$E' = \{(u, v) : u, v \in V \text{ e } u \neq v\}$$

Atribuímos um custo inteiro a cada aresta em E' da seguinte maneira:

$$c(u,v) = \begin{cases} 1 & \text{se } (u,v) \in E, \\ \rho|V|+1 & \text{ou} \end{cases}$$

Podemos criar representações de G' e c a partir da representação de G em tempo polinomial em $|V|$ e $|E|$.

Agora, considere o problema do caixeiro-viajante (G', c). Se o grafo original G tem um ciclo hamiltoniano H, então a função custo c atribui a cada aresta de H um custo 1 e, assim, (G', c) contém um passeio de custo $|V|$. Por outro lado, se G não contém um ciclo hamiltoniano, qualquer passeio de G' deve usar alguma aresta que não está em E. Porém, qualquer passeio que utilize uma aresta que não está em E tem um custo de no mínimo

$$\begin{aligned}(\rho|V|+1)+(|V|-1) &= \rho|V|+|V| \\ &> \rho|V|\end{aligned}$$

Como as arestas que não estão em G são tão dispendiosas, existe uma lacuna de no mínimo $r|V|$ entre o custo de um passeio que é um ciclo hamiltoniano em G (custo $|V|$) e o custo de qualquer outro passeio (custo no mínimo igual a $r|V| + |V|$). Portanto, o custo de um passeio que não é um ciclo hamiltoniano em G é no mínimo um fator $r + 1$ maior do que o custo de um passeio que é um ciclo hamiltoniano em G.

Agora, suponha que apliquemos o algoritmo de aproximação A ao problema do caixeiro-viajante (G', c). Como é garantido que A retorna um passeio de custo não mais do que r vezes o custo de um passeio ótimo, se G contém um ciclo hamiltoniano, então A deve devolvê-lo. Se G não tem nenhum ciclo hamiltoniano, então A retorna um ciclo de custo maior que $r|V|$. Assim, podemos usar A para resolver o problema do ciclo hamiltoniano em tempo polinomial.

A prova do Teorema 35.3 serve como exemplo de uma técnica geral para provar que não podemos aproximar um problema muito bem. Suponha que, dado um problema NP-difícil X, podemos produzir em tempo polinomial um problema de minimização Y tal que instâncias "sim" de X correspondem a instâncias de Y com valor no máximo k (para algum k), mas que instâncias "não" de X correspondam a instâncias de Y com valor maior que rk. Então mostramos que, a menos que P = NP, não existe nenhum algoritmo de aproximação r para problema Y.

Exercícios

35.2-1 Suponha que um grafo não dirigido completo $G = (V, E)$ com no mínimo três vértices tenha uma função custo c que satisfaz a desigualdade triangular. Prove que $c(u, v) \geq 0$ para todo $u, v \in V$.

35.2-2 Mostre como podemos transformar em tempo polinomial uma instância do problema do caixeiro-viajante em outra instância cuja função custo satisfaça a desigualdade triangular. As duas instâncias devem ter o mesmo conjunto de passeios ótimos. Explique por que tal transformação em tempo polinomial não contradiz o Teorema 35.3, considerando P ≠ NP.

35.2-3 Considere a seguinte ***heurística do ponto mais próximo*** para construir um passeio de caixeiro-viajante aproximado. Comece com um ciclo trivial consistindo em um único vértice escolhido arbitrariamente. Em cada etapa, identifique o vértice u que não está no ciclo, mas cuja distância até qualquer vértice no ciclo é mínima. Suponha que, nesse ciclo, o vértice que está mais próximo de u seja o vértice v. Estenda o ciclo para incluir u, inserindo u logo depois de v. Repita até que todos os vértices estejam no ciclo. Prove que essa heurística retorna um passeio cujo custo total não é mais do que duas vezes o custo de um passeio ótimo.

35.2-4 No ***problema do caixeiro-viajante com gargalo*** desejamos determinar o ciclo hamiltoniano que minimiza o custo da aresta de maior custo no ciclo. Considerando que a função custo satisfaz a desigualdade triangular, mostre que existe um algoritmo de aproximação de tempo polinomial com razão de aproximação 3 para esse problema. (*Sugestão*: Mostre recursivamente que podemos visitar todos os nós em uma árvore geradora de gargalo, como vimos no Problema 23-3, exatamente uma vez, fazendo um passeio completo da árvore e saltando nós, mas sem saltar mais de dois nós intermediários consecutivos. Mostre que a aresta de maior custo em uma árvore geradora de gargalo tem um custo que é no máximo o custo da aresta de maior custo em um ciclo hamiltoniano de gargalo.)

35.2-5 Suponha que os vértices para uma instância do problema do caixeiro-viajante sejam pontos no plano e que o custo $c(u, v)$ seja a distância euclidiana entre os pontos u e v. Mostre que um passeio ótimo nunca cruza com ele mesmo.

35.3 O PROBLEMA DE COBERTURA DE CONJUNTOS

O problema de cobertura de conjuntos é um problema de otimização que modela muitos problemas de alocação de recursos. Seu problema de decisão correspondente generaliza o problema NP-completo de cobertura de vértices e, portanto, também é NP-difícil. Porém, o algoritmo de aproximação desenvolvido para tratar o problema de cobertura de vértices não se aplica aqui, então precisamos tentar outras abordagens. Examinaremos uma heurística gulosa simples com uma razão de aproximação logarítmica. Isto é, à medida que o tamanho da instância aumenta, o tamanho da solução aproximada pode crescer em relação ao tamanho de uma solução ótima. Contudo, como a função logaritmo cresce muito lentamente, esse algoritmo de aproximação pode, mesmo assim, dar resultados úteis.

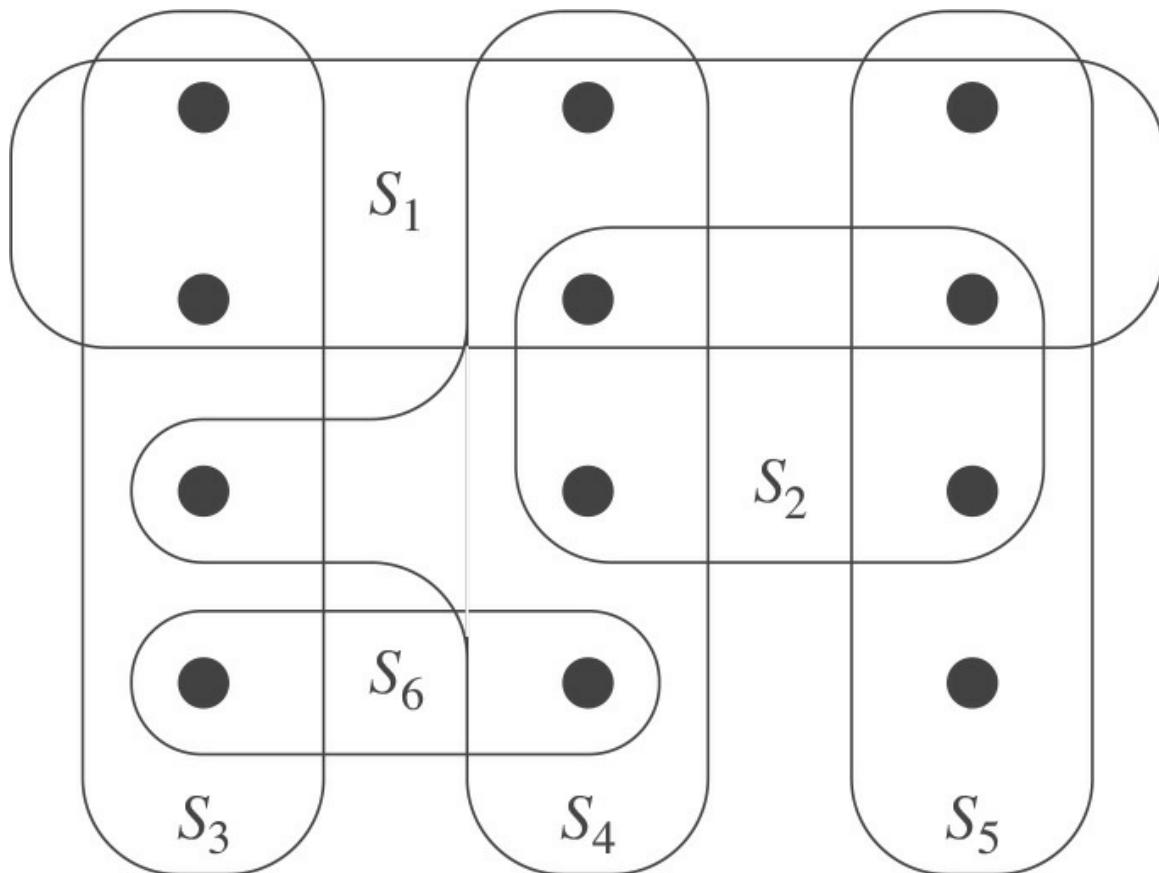
Uma instância (*X*, F) do ***problema de coberta de conjuntos*** consiste em um conjunto finito *X* e uma família F de subconjuntos de *X*, tal que todo elemento de *X* pertence a, no mínimo, um subconjunto em F:

$$X = \bigcup_{S \in \mathfrak{F}} S.$$

Dizemos que um subconjunto $S \in$ F ***cobre*** seus elementos. O problema é encontrar um subconjunto de tamanho mínimo $\subseteq$ F cujos membros cubram todo o *X*:

$$X = \bigcup_{S \in \mathscr{C}} S. \qquad (35.8)$$

Dizemos que qualquer que satisfaça a equação (35.8) ***cobre*** *X*. A Figura 35.3 ilustra o problema de cobertura de conjuntos. O tamanho de C é o número de conjuntos que ele contém, em vez do número de elementos individuais

nesses conjuntos, visto que cada subconjunto de C que cobre *X* deve conter todos os |X| elementos individuais. Na Figura 35.3, a cobertura do conjunto mínimo tem tamanho 3.

O problema de cobertura de conjuntos abstrai muitos problemas combinatórios que surgem comumente. Como um exemplo simples, suponha que *X* represente um conjunto de habilidades necessárias para resolver um problema e que temos um dado conjunto de pessoas disponíveis para trabalhar no problema. Desejamos formar um comitê que contenha o menor número de pessoas possível tal que, para toda habilidade requerida em *X*, exista no mínimo um membro do comitê que tenha essa habilidade. Na versão de decisão do problema de cobertura de conjuntos, perguntamos se existe ou não uma cobertura de tamanho no máximo k, onde k é um parâmetro adicional especificado na instância do problema. A versão de decisão do problema é NP-completa, como o Exercício 35.3-2 pede para mostrar.

Um algoritmo de aproximação guloso

O método guloso funciona escolhendo, em cada fase, o conjunto *S* que cobre o maior número de elementos restantes que não estão cobertos.

GREEDY-SET-COVER(X, $\mathfrak{F}$)
1 $U = X$
2 $\mathscr{C} = \emptyset$
3 **while** $U \neq \emptyset$
4 selecione um $S \in \mathfrak{F}$ que maximiza $|S \cap U|$
5 $U = U - S$
6 $\mathscr{C} = \mathscr{C} \cup |S|$
7 **return** $\mathscr{C}$

Figura 35.3 Uma instância (X, F) do problema de cobertura de conjuntos, onde X consiste nos 12 pontos pretos e F = $(S_1, S_2, S_3, S_4, S_5, S_6\}$. Uma cobertura de conjuntos de tamanho mínimo é = (S_3, S_4, S_5). O algoritmo guloso produz uma cobertura de tamanho 4 selecionando os conjuntos S_1, S_4, S_5 e S_3 ou os conjuntos S_1, S_4, S_5 e S_6, em ordem.

No exemplo da Figura 35.3, Greedy-Set-Cover adiciona a , em ordem, os conjuntos S_1, S_4 e S_5, seguidos por S_3 ou por S_6.

O algoritmo funciona da maneira descrita a seguir. O conjunto U contém, em cada estágio, o conjunto de elementos não cobertos restantes. O conjunto contém a cobertura que está sendo construída. A linha 4 é a etapa de tomada de decisão gulosa, que escolhe um subconjunto S que cobre o maior número possível de elementos não cobertos (decidindo empates arbitrariamente). Depois de S ser selecionado, a linha 5 remove seus elementos U e a linha 6 coloca S em . Quando o algoritmo termina, o conjunto contém uma subfamília de F que cobre X.

É fácil implementar o algoritmo Greedy-Set-Cover para execução em tempo polinomial em $|X|$ e $|F|$. Visto que o número de iterações do laço nas linhas 3–6 é limitado por cima por $\min(|X|, |F|)$ e que podemos implementar o corpo do laço para ser executado no tempo $O(|S||F|)$, uma implementação simples é executada no tempo $O(|S||F| \min(|X|,|F|))$. O Exercício 35.3-3 pede um algoritmo de tempo linear.

Análise

Agora, mostramos que o algoritmo guloso devolve uma cobertura de conjuntos que não é muito maior que uma cobertura de conjuntos ótima. Por conveniência, neste capítulo denotaremos por $H(d)$ o d-ésimo número harmônico $\sum_{i=1}^{d} 1/i$ (consulte a Seção A.1). Como condição de contorno, definimos $H(0) = 0$.

Teorema 35.4

Greedy-Set-Cover é um algoritmo polinomial de $r(n)$-aproximação, onde

$$\rho(n) = H(\max\{|S| : S \in \mathfrak{F}\}) .$$

Prova Já mostramos que Greedy-Set-Cover é executado em tempo polinomial.

Para mostrar que Greedy-Set-Cover é um algoritmo de $r(n)$-aproximação atribuímos um custo 1 a cada conjunto selecionado pelo algoritmo, distribuímos esse custo pelos elementos cobertos pela primeira vez e depois usamos esses custos para deduzir a relação desejada entre o tamanho de uma cobertura de conjunto ótima * e o tamanho da cobertura de conjunto C devolvida pelo algoritmo. Seja S_i o i-ésimo subconjunto selecionado por Greedy-Set-Cover; o algoritmo incorre em um custo de 1 quando acrescenta S_i a . Desdobramos esse custo da seleção de S_i uniformemente entre os elementos cobertos pela primeira vez por S_i . Seja c_x o custo alocado ao elemento x, para cada $x \in X$. O custo assim desdobrado é atribuído a cada elemento somente uma vez, quando ele é coberto pela primeira vez. Se x é coberto pela primeira vez por S_i, então

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$

Cada etapa do algoritmo atribui uma unidade de custo e, portanto,

$$|\mathscr{C}| = \sum_{x \in X} c_x . \qquad (35.9)$$

Cada elemento $x \in X$ está em, no mínimo, um conjunto da cobertura ótima C*, portanto temos

$$\sum_{S\in\mathscr{C}^*}\sum_{x\in S} c_x \geq \sum_{x\in X} c_x. \qquad (35.10)$$

Combinando a equação (35.9) e a desigualdade (35.10), temos que

$$|\mathscr{C}| \leq \sum_{S\in\mathscr{C}^*}\sum_{x\in S} c_x. \qquad (35.11)$$

O restante da prova se baseia na seguinte desigualdade fundamental, que provaremos em breve. Para qualquer S que pertença à família F,

$$\sum_{x\in S} c_x \leq H(|S|). \qquad (35.12)$$

Das desigualdades (35.11) e (35.12), decorre que

$$\begin{aligned} |\mathscr{C}| &\leq \sum_{S\in\mathscr{C}^*} H(|S|) \\ &\leq |\mathscr{C}^*| \cdot H(\max\{|S| : S \in \mathfrak{F}\}) \end{aligned}$$

o que prova o teorema.

Resta apenas provar a desigualdade (35.12). Considere qualquer conjunto $S \in$ F e qualquer $i = 1, 2, ..., ||$, e seja

$$u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$$

o número de elementos em S que permanecem não cobertos depois de o algoritmo ter selecionado os conjuntos S_1, S_2, ..., S_i. Definimos $u_0 = |S|$ como o número de elementos de S que estão inicialmente não cobertos. Seja k o índice mínimo tal que $u_k = 0$, de modo que cada elemento em S é coberto por no mínimo um dos conjuntos S_1, S_2, ..., S_k e algum elemento em S não é coberto por $S_1 \cup S_2 \cup \ldots \cup S_{k-1}$. Então, $u_{i-1} \geq u_i$, e $u_{i-1} - u_i$ elementos de S são cobertos pela primeira vez por S_i, para $i = 1, 2, ..., k$. Assim,

$$\sum_{x\in S} c_x = \sum_{i=1} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})}.$$

Observe que

$$\begin{aligned} |S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| &\geq |S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \\ &= u_{i-1}, \end{aligned}$$

porque a escolha gulosa de S_i garante que S não pode cobrir mais elementos novos que S_i (caso contrário, o algoritmo teria escolhido S em vez de S_i). Consequentemente, obtemos

$$\sum_{x\in S} c_x = \sum_{i=1} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}.$$

Agora, limitamos essa quantidade da seguinte maneira:

$$
\begin{aligned}
\sum_{x\in S} c_x &\le \sum_{i=1} (u_{i-1} - u_i)\cdot \frac{1}{u_{i-1}}.\\
&= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}}\\
&\le \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} && (\text{porque } j \le u_{i-1})\\
&= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right)\\
&= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i))\\
&= H(u_0) - H(u_k) && (\text{porque a soma é telescópica})\\
&= H(u_0) - H(0)\\
&= H(u_0) && (\text{porque } H(0) = 0)\\
&= H(|S|),
\end{aligned}
$$

o que conclui a prova da desigualdade (35.12).

Corolário 35.5

Greedy-Set-Cover é um algoritmo polinomial de $(\ln |X|+1)$-aproximação.

Prova Use a desigualdade (A.14) e o Teorema 35.4.

Em algumas aplicações, $\max\{|S| : S \in F\}$ é uma constante pequena e, assim, a solução devolvida por Greedy-Set-Cover é no máximo um multiplo constante pequeno, maior que a ótima. Uma aplicação desse tipo ocorre quando essa heurística encontra uma cobertura de vértices aproximada para um grafo cujos vértices têm no máximo grau 3. Nesse caso, a solução encontrada por Greedy-Set-Cover não é mais do que $H(3) = 11/6$ vezes maior do que uma solução ótima, o que garante um desempenho um pouco melhor do que o de Approx-Vertex-Cover.

Exercícios

35.3-1 Considere cada uma das seguintes palavras um conjunto de letras: {arid, dash, drain, heard, lost, nose, shun, slate, snare, thread}. Mostre qual cobertura de conjuntos Greedy-Set-Cover produz quando os empates são quebrados em favor da palavra que aparece em primeiro lugar no dicionário.

35.3-2 Mostre que a versão de decisão do problema de cobertura de conjuntos é NP-completa, reduzindo o problema de cobertura de vértices.

35.3-3 Mostre como implementar Greedy-Set-Cover de tal maneira que ele seja executado no tempo $O\left(\sum_{S\in F}|S|\right)$.

35.3-4 Mostre que a forma mais fraca do Teorema 35.4 a seguir é trivialmente verdadeira:

$$|\mathcal{C}| \le |\mathcal{C}^*| \max\{|S| : S \in \mathfrak{F}\}.$$

35.3-5 Greedy-Set-Cover pode devolver várias soluções diferentes, dependendo de como decidimos os empates na linha 4. Dê um procedimento Bad-Set-Cover-Instance(n) que devolva uma instância de n elementos do problema de cobertura de conjuntos para a qual, dependendo de como decidimos os empates na linha 4,

Greedy-Set-Cover pode devolver uma quantidade de soluções diferentes que é exponencial em n.

35.4 ALEATORIZAÇÃO E PROGRAMAÇÃO LINEAR

Nesta seção, estudaremos duas técnicas úteis para o projeto de algoritmos de aproximação: aleatorização e programação linear. Apresentaremos um algoritmo aleatorizado simples para uma versão de otimização de satisfazibilidade 3-CNF e depois usaremos programação linear para ajudar a projetar um algoritmo de aproximação para uma versão ponderada do problema de cobertura de vértices. Esta seção apenas toca na superfície dessas duas técnicas eficientes. As notas do capítulo apresentam referências para estudo adicional dessas áreas.

Um algoritmo de aproximação aleatorizado para satisfazibilidade MAX-3-CNF

Exatamente como os algoritmos aleatorizados calculam soluções exatas, alguns algoritmos aleatorizados calculam soluções aproximadas. Dizemos que um algoritmo aleatorizado para um problema tem uma ***razão de aproximação*** $r(n)$ se, para qualquer entrada de tamanho n, o custo *esperado* C da solução produzida pelo algoritmo aleatorizado está a menos de um fator $r(n)$ do custo C^* de uma solução ótima:

$$máx\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n) \tag{35.13}$$

Denominamos um algoritmo aleatorizado que consegue uma razão de aproximação $r(n)$ ***algoritmo aleatorizado de*** $r(n)$***-aproximação*** $r(n)$. Em outras palavras, um algoritmo de aproximação aleatorizado é semelhante a um algoritmo de aproximação determinístico, exceto que a razão de aproximação é para o custo esperado.

Uma instância específica de satisfazibilidade 3-CNF, definida na Seção 34.4, pode ou não ser satisfazível. Para ser satisfazível, é preciso que exista uma atribuição das variáveis, de modo que toda cláusula tenha valor 1. Se uma instância não é satisfazível, seria interessante calcular quão "próxima" de satisfazível ela está, isto é, determinar uma atribuição das variáveis que satisfaça o maior número possível de cláusulas. Denominamos o problema de maximização resultante ***satisfazibilidade MAX-3-CNF***. A entrada para a satisfazibilidade MAX-3-CNF é a mesma da satisfazibilidade 3-CNF, e o objetivo é retornar uma atribuição das variáveis que maximize o número de cláusulas de valor 1. Agora mostramos que atribuir aleatoriamente cada variável com 1 com probabilidade 1/2 e com 0 com probabilidade 1/2 resulta em algoritmo aleatorizado de aproximação 8/7. De acordo com a definição de satisfazibilidade 3-CNF da Seção 34.4, cada cláusula deve consistir em exatamente três literais distintos. Além disso, supomos que nenhuma cláusula contém, ao mesmo tempo, uma variável e sua negação (o Exercício 35.4-1 pede para eliminar esta última hipótese).

Teorema 35.6

Dada uma instância de satisfazibilidade MAX-3-CNF com n variáveis $x_1, x_2, ..., x_n$ e m cláusulas, o algoritmo aleatorizado que atribui independentemente cada variável com 1 com probabilidade 1/2 e com 0 com probabilidade 1/2 é um algoritmo aleatorizado de razão de aproximação 8/7.

Prova Suponha que atribuímos independentemente cada variável com 1 com probabilidade 1/2 e com 0 com probabilidade 1/2. Para $i = 1, 2, ..., m$, definimos a variável aleatória indicadora

Y_i = I{a cláusula i é satisfeita} ,

de modo que $Y_i = 1$ desde que tenhamos atribuído no mínimo um dos literais na i-ésima cláusula com 1. Visto que nenhum literal aparece mais de uma vez na mesma cláusula, e como supomos que nenhuma variável e sua negação aparecem ao mesmo tempo na mesma cláusula, as configurações dos três literais em cada cláusula são independentes. Uma cláusula não é satisfeita somente se todos os seus três literais são atribuídos com 0 e, assim, Pr{cláusula i não é

satisfeita} = $(1/2)^3 = 1/8$. Assim, Pr{cláusula i é satisfeita } = 1 – 1/8 = 7/8. Então, pelo Lema 5.1, temos $E[Y_i] = 7/8$. Seja Y o número global de cláusulas satisfeitas, de modo que $Y = Y_1 + Y_2 + \ldots + Y_m$. Então, temos

$$\begin{aligned} E[Y] &= E\left[\sum_{i=1}^{m} Y_i\right] \\ &= \sum_{i=1}^{m} E[Y_i] \qquad \text{(por linearidade de esperança)} \\ &= \sum_{i=1}^{m} 7/8 \\ &= 7m/8 \end{aligned}$$

É claro que m é um limite superior para o número de cláusulas satisfeitas e, consequentemente, a razão de aproximação é no máximo $m/(7m/8) = 8/7$.

Aproximação de cobertura de vértices ponderada utilizando programação linear

No ***problema de cobertura de vértices de peso mínimo***, temos um grafo não dirigido $G = (V, E)$ no qual cada vértice $v \in V$ tem um peso positivo associado $w(v)$. Para qualquer cobertura de vértices $V' \subseteq V$, definimos o peso da cobertura de vértices $w(V') = \sum_{v \in V'} w(v)$. A meta é encontrar uma cobertura de vértices de peso mínimo.

Não podemos aplicar o algoritmo usado para cobertura de vértices não ponderada nem podemos usar uma solução aleatória; ambos os métodos podem retornar soluções que estão longe de ótimas. Porém, calcularemos um limite inferior para o peso da cobertura de vértices de peso mínimo usando um programa linear. Então, "arredondaremos" essa solução e a usaremos para obter uma cobertura de vértices.

Suponha que associamos uma variável (v) a cada vértice $v \in V$ e que definimos $x(v)$ igual a 0 ou a 1 para cada $v \in V$. Inserimos v na cobertura de vértices se e somente se $x(v) = 1$. Então, podemos escrever a seguinte restrição: para qualquer aresta (u, v), no mínimo um dentre u e v deve estar na cobertura de vértices como $x(u) + x(v) \geq 1$. Essa visão dá origem ao seguinte ***pro- grama inteiro 0-1*** para encontrar uma cobertura de vértices de peso mínimo:

$$\text{minimizar} \quad \sum_{v \in V} w(v)x(v) \qquad (35.14)$$

sujeito a

$$\begin{aligned} x(u) + x(v) &\geq 1 && \text{para cada } (u, v) \in E && (35.15) \\ x(v) &\in \{0, 1\} && \text{para cada } (u, v) \in V. && (35.16) \end{aligned}$$

No caso especial em que todo os pesos $w(v)$ são iguais a 1, essa formulação é a versão de otimização do problema de cobertura de vértices NP difícil. Contudo, suponha que eliminamos a restrição $x(v) \in \{0, 1\}$ e a substituímos por $0 \leq x(v) \leq 1$. Então, obtemos o seguinte programa linear, conhecido como ***relaxação linear***:

$$\text{minimizar} \quad \sum_{v \in V} w(v)x(v) \qquad (35.17)$$

sujeito a

$$\begin{aligned} x(u) + x(v) &\geq 1 && \text{para cada } v \in E && (35.18) \\ x(v) &\leq 1 && \text{para cada } v \in V && (35.19) \\ x(v) &\geq 0 && \text{para cada } v \in V\,. && (35.20) \end{aligned}$$

Qualquer solução viável para o programa inteiro 0-1 nas linhas (35.14)–(35.16) também é uma solução viável para o programa linear nas linhas (35.17)–(35.20). Portanto, o valor de uma solução ótima para o programa linear dá um

limite inferior para o valor de uma solução ótima para o programa inteiro 0-1 e, por consequência, um limite inferior para o peso ótimo no problema de cobertura de vértices de peso mínimo.

O procedimento a seguir usa a solução para a relaxação linear para construir uma solução aproximada para o problema de cobertura de vértices de peso mínimo:

```
APPROX-MIN-WEIGHT-VC(G, w)
1  C = ∅
2  calcular x̄, uma solução ótima para o programa linear nas linhas (35.17)–(35.20)
3  for cada v ∈ V
4       if x̄ (v) ≥ 1/2
5            C = C ∪ {v}
6  return C
```

O procedimento Approx-Min-Weight-VC funciona da seguinte maneira: a linha 1 inicializa a cobertura de vértices como vazia. A linha 2 formula o programa linear nas linhas (35.17)— 35.20) e depois resolve esse programa linear. Uma solução ótima dá a cada vértice v um valor associado $x\ (v)$, onde $0 \leq x\ (v) \leq 1$. Usamos esse valor para orientar a escolha dos vértices a adicionar à cobertura de vértices C nas linhas 3–5. Se $x\ (v) \geq 1/2$, adicionamos v a C; caso contrário, não adicionamos. Na realidade, estamos "arredondando" cada variável fracionária na solução do programa linear para 0 ou 1, de modo a obter uma solução para o programa inteiro 0-1 nas linhas (35.14)–(35.16). Finalmente, a linha 6 devolve a cobertura de vértices C.

Teorema 35.7

O algoritmo Approx-Min-Weight-VC é um algoritmo polinomial de 2-aproximação para o problema de cobertura de vértices de peso mínimo.

Prova Como existe um algoritmo de tempo polinomial para resolver o programa linear na linha 2 e como o laço **for** das linhas 3–5 é executado em tempo polinomial, Approx-Min-Weight--VC é um algoritmo de tempo polinomial.

Agora mostramos que Approx-Min-Weight-VC é um algoritmo de 2-aproximação. Seja C^* uma solução ótima para o problema de cobertura de vértices de peso mínimo, e seja z^* o valor de uma solução ótima para o programa linear nas linhas (35.17)–(35.20). Visto que uma cobertura de vértices ótima é uma solução viável para o programa linear, z^* deve ser um limite inferior para $w(C^*)$, isto é,

$$z^* \leq w(C^*) . \tag{35.21}$$

Em seguida, afirmamos que, arredondando os valores fracionários das variáveis $x(v)$, produzimos um conjunto C que é uma cobertura de vértices e satisfaz $w(C) \leq 2z_*$. Para ver que C é uma cobertura de vértices, considere qualquer aresta $(u, v) \in E$. Pela restrição (35.18), sabemos que $x(u) + x(v) \geq 1$, o que implica que pelo menos um dentre $x(u)$ e $x(v)$ é no mínimo 1/2. Portanto, no mínimo um dentre u e v é incluído na cobertura de vértices, e então toda a aresta estará coberta.

Agora consideramos o peso da cobertura. Temos

$$\begin{aligned}
z^* &= \sum_{v \in V} w(v)\bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v)\bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\
&= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\
&= \frac{1}{2}\sum_{v \in C} w(v) \\
&= \frac{1}{2} w(C). \qquad (35.22)
\end{aligned}$$

Combinando as desigualdades (35.21) e (35.22) obtemos

$$w(C) \leq 2z^* \leq 2w(C^*) \ ,$$

e, consequentemente, Approx-Min-Weight-VC é um algoritmo de 2-aproximação.

Exercícios

35.4-1 Mostre que, mesmo se permitirmos que uma cláusula contenha ao mesmo tempo uma variável e sua negação, definir uma variável aleatoriamente como 1 com probabilidade 1/2 e como 0 com probabilidade 1/2 ainda resulta em um algoritmo aleatorizado de 8/7-aproximação.

35.4-2 O ***problema de satisfazibilidade MAX-CNF*** é semelhante ao problema de satisfazibilidade MAX-3-CNF, exceto por não restringir que cada cláusula tenha exatamente três literais. Dê um algoritmo aleatorizado de 2-aproximação para o problema de satisfazibilidade MAX-CNF.

35.4-3 No problema MAX-CUT, temos um grafo não dirigido não ponderado $G = (V, E)$. Definimos um corte $(S, V - S)$ como no Capítulo 23 e o ***peso*** de um corte como o número de arestas que cruzam o corte. A meta é encontrar um corte de peso máximo. Suponha que, para cada vértice v inserimos, aleatória e independentemente, v em S com probabilidade 1/2 e em $V - S$ com probabilidade 1/2. Mostre que esse algoritmo é um algoritmo aleatorizado de 2-aproximação.

35.4-4 Mostre que as restrições na linha (35.19) são redundantes no sentido de que, se as eliminarmos do programa linear das linhas (35.17)–(35.20), qualquer solução ótima para o programa linear resultante deve satisfazer $x(v) \leq 1$ para cada $v \in V$.

35.5 O PROBLEMA DA SOMA DE SUBCONJUNTOS

Lembre-se de que dissemos na Seção 34.5.5, que uma instância do problema de soma de subconjuntos é um par (S, t), onde S é um conjunto $\{x_1, x_2,..., x_n\}$ de inteiros positivos e t é um inteiro positivo. Esse problema de decisão pergunta se existe um subconjunto de S cuja soma seja exatamente o valor alvo t. Como vimos na Seção 34.5.5, esse problema é NP-completo.

O problema de otimização associado a esse problema de decisão surge em aplicações práticas. No problema de otimização, desejamos encontrar um subconjunto de $\{x_1, x_2,..., x_n\}$ cuja soma seja a maior possível, mas não maior que t. Por exemplo, podemos ter um caminhão que não pode transportar mais de t quilogramas e n caixas diferentes para

despachar, das quais a i-ésima caixa pesa x_i quilogramas. Desejamos encher o caminhão com a carga mais pesada possível, sem exceder o limite de peso dado.

Nesta seção, apresentamos um algoritmo de tempo exponencial que calcula o valor ótimo para esse problema de otimização e depois mostramos como modificar o algoritmo de modo que ele se torne um esquema de aproximação de tempo completamente polinomial. (Lembre-se de que um esquema de aproximação de tempo completamente polinomial tem um tempo de execução que é polinomial em $1/e$, bem como no tamanho da entrada.)

Um algoritmo de tempo exponencial exato

Suponha que calculamos, para cada subconjunto S' de S, a soma dos elementos em S' e então selecionamos, entre os subconjuntos cuja soma não excede t, aquele cuja soma seja a mais próxima de t. É claro que esse algoritmo devolveria a solução ótima, mas ele poderia demorar tempo exponencial. Para implementar esse algoritmo, poderíamos utilizar um procedimento iterativo que, na iteração i, calcula as somas de todos os subconjuntos de $\{x_1, x_2,..., x_i\}$ usando como ponto de partida as somas de todos os subconjuntos de $\{x_1, x_2,..., x_i{-1}\}$. Ao fazer isso, perceberíamos que tão logo um determinado subconjunto S' tivesse uma soma maior do que t, não haveria nenhuma razão para mantê-lo, já que nenhum superconjunto de S' poderia ser a solução ótima. Agora damos uma implementação dessa estratégia.

O procedimento Exact-Subset-Sum toma um conjunto de entrada $S = \{x_1, x_2,..., x_n\}$ e um valor alvo t. Veremos seu pseudocódigo em breve. Esse procedimento calcula iterativamente L_i, a lista de somas de todos os subconjuntos de $\{x_1,..., x_i\}$ que não excedem t e depois devolve o valor máximo em L_n.

Se L é uma lista de inteiros positivos e x é um outro inteiro positivo, seja $L + x$ a lista de inteiros derivados de L somando x a cada elemento de L. Por exemplo, se $L = \langle 1, 2, 3, 5, 9\rangle$, então $L + 2 = \langle 3, 4, 5, 7, 11\rangle$. Também usamos essa notação para conjuntos, de modo que

$$S + x = \{s + x : s \in S\} .$$

Usamos também um procedimento auxiliar Merge-Lists(L, L') que retorna a lista ordenada, que é o resultado da intercalação de suas duas listas ordenadas de entrada L e L', após a remoção de valores duplicados. Como o procedimento Merge que usamos na ordenação por intercalação (Seção 2.3.1), Merge-Lists é executado no tempo $O(|L| + |L'|)$. (Omitimos o pseudocódigo para

Exact-Subset-Sum(S, t)

1 $n = |S|$
2 $L_0 = \langle 0\rangle$
3 **for** $i = 1$ to n
4 $\quad L_i =$ Merge-Lists$(L_{i-1}, L_{i-1} + x_i)$
5 $\quad$ elimine de L_i todo elemento maior que t
6 **return** o maior elemento em L_n

Para ver como Exact-Subset-Sum funciona, seja P_i o conjunto de todos os valores que podem ser obtidos pela seleção de um subconjunto (possivelmente vazio) de $\{x_1, x_2,..., x_i\}$ e adição de seus elementos. Por exemplo, se $S = \{1, 4, 5\}$, então

$$P_1 = \{0, 1\} ,$$
$$P_2 = \{0, 1, 4, 5\} ,$$
$$P_3 = \{0, 1, 4, 5, 6, 9, 10\} .$$

Dada a identidade

$$P_i = P_{i-1} \cup (P_{i-1} + x_i), \tag{35.23}$$

podemos provar por indução em i (veja o Exercício 35.5-1) que a lista L_i é uma lista ordenada que contém cada elemento de P_i cujo valor não é maior que t. Visto que o comprimento de L_i pode ser até 2^i, Exact-Subset-Sum é um algoritmo de tempo exponencial em geral, embora seja um algoritmo de tempo polinomial nos casos especiais em que t é polinomial em $|S|$ ou todos os números em S são limitados por um polinômio em $|S|$.

Um esquema de aproximação de tempo completamente polinomial

Podemos derivar um esquema de aproximação de tempo completamente polinomial para o problema da soma de subconjuntos "desbastando" cada lista L_i depois de criada. A ideia que fundamenta o desbaste é que, se dois valores em L estão próximos um do outro, então, visto que o que queremos é uma solução aproximada, não precisamos manter ambos explicitamente. Mais exatamente, usamos um parâmetro de desbaste d tal que $0 < d < 1$. Quando ***desbastamos*** uma lista L usando o parâmetro de desbaste d, eliminamos o maior número possível de elementos de L, de modo tal que, se L' é o resultado do desbaste de L, então para cada elemento y que foi eliminado de L ainda existe um elemento z em L' que se aproxima de y, isto é,

$$\frac{y}{1+\delta} \leq z \leq y. \tag{35.24}$$

Podemos pensar em z como um "representante" de y na nova lista L'. Cada elemento y eliminado é representado por um elemento z restante que satisfaz a desigualdade (35.24). Por exemplo, se $d = 0{,}1$ e

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \ ,$$

então podemos desbastar L para obter

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle \ ,$$

onde o valor eliminado 11 é representado por 10, os valores eliminados 21 e 22 são representados por 20, e o valor eliminado 24 é representado por 23. Como todo elemento da versão desbastada da lista também é um elemento da versão original da lista, o desbaste pode reduzir drasticamente o número de elementos mantidos na lista, ao mesmo tempo que mantém um valor representativo próximo (e ligeiramente menor) na lista para cada elemento eliminado.

O procedimento a seguir desbasta a lista $L = \langle y_1, y_2, .., y_m \rangle$ no tempo $Q(m)$, dados L e d e supondo que L está ordenada em sequência monotonicamente crescente. A saída do procedimento é uma lista desbastada e ordenada.

```
TRIM(L, δ)
1   seja m o comprimento de L
2   L' = ⟨y_1⟩
3   último = y_1
4   for i = 2 to m
5       if y_i > último · (1 + δ)          // y_i ≥ último porque L está ordenada
6           anexar y_i ao final de L'
7               último = y_i
8   return L'
```

O procedimento examina os elementos de L em ordem monotonicamente crescente. Um número é anexado à lista retornada L' somente se ele é o primeiro elemento de L ou se não pode ser representado pelo número mais recente inserido em L'.

Dado o procedimento Trim, podemos construir nosso esquema de aproximação da maneira descrita a seguir. Esse procedimento toma como entrada um conjunto $S = \{x_1, x_2, ..., x_n\}$ de n inteiros (em ordem arbitrária), um inteiro alvo t e um "parâmetro de aproximação", e onde

$$0 < \epsilon < 1 . \tag{35.25}$$

Devolve um valor z cujo valor encontra-se a menos de um fator $1 + e$ da solução ótima.

Approx-Subset-Sum(S, t, ϵ)
1 $n = |S|$
2 $L_0 = \langle 0 \rangle$
3 **for** $i = 1$ **to** n
4 $L_i =$ Merge-Lists$(L_{i-1}, L_{i-1} + x_i)$
5 $L_i =$ Trim$(L_i, \epsilon/2n)$
6 elimine de L_i todo elemento maior que t
7 seja z^* o maior valor em L_n
8 **return** z^*

A linha 2 inicializa a lista L_0 como a lista que contém apenas o elemento 0. O laço **for** das linhas 3–6 calcula L_i como uma lista ordenada que contém uma versão adequadamente desbastada do conjunto P_i, eliminados todos os elementos maiores que t . Como L_i é criada a partir de L_{i-1}, temos de assegurar que o desbaste repetido não introduza imprecisão composta excessiva. Mais adiante veremos que Approx-Subset-Sum devolve uma aproximação correta, se existir.

Como exemplo, suponha que tenhamos a instância

$$S = \langle 104, 102, 201, 101 \rangle$$

com $t = 308$ e $e = 0{,}40$. O parâmetro de desbaste d é $e/8 = 0{,}05$. Approx-Subset-Sum calcula os valores a seguir nas linhas indicadas:

linha 2: $L_0 = \langle 0 \rangle$,

linha 4: $L_1 = \langle 0, 104 \rangle$,
linha 5: $L_1 = \langle 0, 104 \rangle$,
linha 6: $L_1 = \langle 0, 104 \rangle$,

linha 4: $L_2 = \langle 0, 102, 104, 206 \rangle$,
linha 5: $L_2 = \langle 0, 102, 206 \rangle$,

linha 6: $L_2 = \langle 0, 102, 206 \rangle$,

linha 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$,
linha 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$,
linha 6: $L_3 = \langle 0, 102, 201, 303 \rangle$,

linha 4: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,
linha 5: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$,
linha 6: $L_4 = \langle 0, 101, 201, 302 \rangle$.

O algoritmo devolve $z^* = 302$ como sua resposta, que se encontra a bem menos de $e = 40\%$ da resposta ótima $307 = 104 + 102 + 101$; de fato, ela está a menos de 2%.

Teorema 35.8

Approx-Subset-Sum é um esquema de aproximação de tempo completamente polinomial para o problema da soma de subconjuntos.

Prova As operações de desbaste L_i na linha 5 e de eliminação em L_i de todo elemento maior que t mantêm a seguinte propriedade: todo elemento de L_i também é um elemento de P_i. Portanto, o valor z devolvido na linha 8 é de fato a soma de algum subconjunto de S. Seja $y^* \in P_n$ uma solução ótima para o problema da soma de subconjuntos. Então, pela linha 6, sabemos que $z^* \leq y^*$. Pela desigualdade (35.1), precisamos mostrar que $y^*/z^* \leq 1 + e$. Devemos também mostrar que o algoritmo é executado em tempo polinomial, tanto em $1/e$ quanto no tamanho da entrada.

O Exercício 35.5.-2 pede que você mostre que, para todo elemento y em P_i que é no máximo t, existe um elemento $z \in L_i$ tal que

$$\frac{y}{(1+\epsilon/2n)^i} \leq z \leq y. \tag{35.26}$$

A desigualdade (35.26) deve ser válida para $y^* \in P_n$ e, portanto, há em elemento $z \in L_n$ tal que

$$\frac{y^*}{(1+\epsilon/2n)^n} \leq z \leq y^*.$$

e, assim,

$$\frac{y^*}{z} \leq \left(1+\frac{\epsilon}{2n}\right)^n. \tag{35.27}$$

Visto que existe um $z \in L_n$ que satisfaz a desigualdade (35.27), a desigualdade deve ser válida para z^*, que é o maior valor em L_n; isto é,

$$\frac{y^*}{z^*} \leq \left(1+\frac{\epsilon}{2n}\right)^n. \tag{35.28}$$

Agora, mostramos que y^*/z^* 1 , mostrando que $(1\ \ 2n)_n$ 1 . Pela equação (3.14), temos $\lim_{n\to\infty}(1\ /\ 2n)_n\ e/2$. O Exercício 35.5-3 lhe pede para mostrar que

$$\frac{d}{dn}\left(1+\frac{\epsilon}{2n}\right)^n > 0. \tag{35.29}$$

Portanto, a função $(1 + e/2n)_n$ cresce com n à medida que tende a seu limite de $e^{e/2}$, e temos

$$\begin{aligned}\left(1+\frac{\epsilon}{2n}\right)^n &\leq e^{\epsilon/2} \\ &\leq 1+\epsilon/2+(\epsilon/2)^2 && \text{(pela desigualdade (3.13))} \\ &\leq 1+\epsilon && \text{(pela desigualdade (35.25))}\,. \end{aligned} \tag{35.30}$$

A combinação das desigualdades (35.28) e (35.30) conclui a análise da razão de aproximação.

Para mostrar que Approx-Subset-Sum é um esquema de aproximação de tempo completamente polinomial, deduzimos um limite para o comprimento de L_i. Após o desbaste, a relação entre os elementos consecutivos z e z' de L_i deve ser $z'/z > 1 + e/2n$. Isto é, a diferença entre eles deve ser de no mínimo um fator $1 + e/2n$. Portanto, cada lista contém o valor 0, possivelmente o valor 1, e até $\log_{1 + e/2n} t$ valores adicionais. O número de elementos em cada lista L_i é no máximo

$$\begin{aligned}\log_{1+\epsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\epsilon/2n)} + 2 \\ &\leq \frac{2n(1+\epsilon/2n)\ln t}{\epsilon} + 2 \qquad \text{(pela desigualdade (3.17))} \\ &< \frac{3n \ln t}{\epsilon} + 2 \qquad \text{(pela desigualdade (35.25))}\ .\end{aligned}$$

Esse limite é polinomial no tamanho da entrada — que é o número de bits $\lg t$ necessários para representar t, mais o número de bits necessários para representar o conjunto S, que é por sua vez polinomial em $n - e$ em $1/e$. Visto que o tempo de execução de Approx-Subset-Sum é polinomial no comprimento das L_i, concluímos que Approx-Subset-Sum é um esquema de aproximação de tempo completamente polinomial.

Exercícios

35.5-1 Prove a equação (35.23), depois mostre que, após a execução da linha 5 de Exact-Subset-Sum, L_i é uma lista ordenada que contém todo elemento de P_i cujo valor não é maior que t.

35.5-2 Usando indução em i, prove a desigualdade (35.26).

35.5-3 Prove a desigualdade (35.29).

35.5-4 Como você modificaria o esquema de aproximação apresentado nesta seção para determinar uma boa aproximação para o menor valor não menor que t que é uma soma de algum subconjunto da lista de entrada dada?

35.5-5 Modifique o procedimento Approx-Subset-Sum para também devolver o subconjunto de S cuja soma é o valor *.

Problemas

35-1 ***Empacotamento em caixas***

Suponha que tenhamos um conjunto de n objetos, onde o tamanho s_i do i-ésimo objeto satisfaz $0 < s_i < 1$. Desejamos empacotar todos os objetos no número mínimo de caixas de tamanho unitário. Cada caixa pod)e conter qualquer subconjunto dos objetos cujo tamanho total não excede 1.

a. Prove que o problema de determinar o número mínimo de caixas exigidas é NP-difícil. (*Sugestão*: Reduza a partir do problema da soma de subconjuntos.)

A heurística do ***primeiro que couber (fisrt-fit)*** toma cada objeto por sua vez e o coloca na primeira caixa que possa acomodá-lo. Seja $S = \sum_{i=1}^{n} s_i$.

b. Demonstre que o número ótimo de caixas necessárias é no mínimo S.

c. Demonstre que a heurística do primeiro que couber deixa no máximo uma caixa cheia até menos da metade.

d. Prove que o número de caixas usadas pela heurística do primeiro que couber nunca é maior que $2S$.

e. Prove uma razão de aproximação 2 para a heurística do primeiro que couber.

f. Dê uma implementação eficiente da heurística do primeiro que couber e analise seu tempo de execução.

35-2 Aproximação do tamanho de um clique máximo

Seja $G = (V, E)$ um grafo não dirigido. Para qualquer $k \geq 1$ defina $G_{(k)}$ como o grafo não dirigido $(V_{(k)}, E_{(k)})$, onde $V_{(k)}$ é o conjunto de todas as k-tuplas ordenadas de vértices de V, e $E_{(k)}$ é definido de modo que $(v_1, v_2,...,v_k)$ seja adjacente a $(w_1, w_2,...,w_k)$ se e somente se, para algum i, o vértice v_i é adjacente a w_i em G ou, então, $v_i = w_i$.

a. Prove que o tamanho do clique máximo em $G_{(k)}$ é igual à k-ésima potência do tamanho do clique máximo em G.

b. Demonstre que, se existe um algoritmo de aproximação que tenha uma razão de aproximação constante para determinar um clique de tamanho máximo, então existe um esquema de aproximação de tempo completamente polinomial para o problema.

35-3 Problema de cobertura de conjuntos ponderada

Suponha que generalizemos o problema de cobertura de conjuntos de modo que cada conjunto S_i na família F tenha um peso associado w_i e o peso de uma cobertura seja $\sum_{S_i \in} w_i$. Desejamos determinar uma cobertura de peso mínimo. (A Seção 35.3 trata o caso em que $w_i = 1$ para todo i.)

Mostre como generalizar a heurística gulosa de cobertura de conjuntos de maneira natural para dar uma solução aproximada para qualquer instância do problema de cobertura de conjuntos ponderada. Mostre que sua heurística tem uma razão de aproximação $H(d)$, onde d é o tamanho máximo de qualquer conjunto S_i.

35-4 Emparelhamento maximal

Lembre-se de que, para um grafo não dirigido G, um emparelhamento é um conjunto de arestas tal que não haja duas arestas no conjunto incidentes no mesmo vértice. Na Seção 26.3, vimos como determinar um emparelhamento máximo em um grafo bipartido. Neste problema, examinaremos emparelhamentos em grafos não dirigidos em geral (isto é, não se exige que os grafos sejam bipartidos).

a. Um ***emparelhamento maximal*** é um emparelhamento que não é um subconjunto próprio de qualquer outro emparelhamento. Mostre que um emparelhamento maximal não precisa ser um emparelhamento máximo exibindo um grafo não dirigido G e um emparelhamento maximal M em G que não seja um emparelhamento máximo. (*Sugestão:* Você pode encontrar um tal grafo com apenas quatro vértices.)

b. Considere um grafo não dirigido $G = (V, E)$. Dê um algoritmo guloso de tempo $O(E)$ para encontrar um emparelhamento maximal em G.

Neste problema, concentraremos nossa atenção em um algoritmo de aproximação de tempo polinomial para emparelhamento máximo. Enquanto o algoritmo mais rápido conhecido para emparelhamento máximo demora tempo superlinear (mas polinomial), o algoritmo de aproximação aqui apresentado será executado em tempo

linear. Você mostrará que o algoritmo guloso de tempo linear para emparelhamento máximo na parte (b) é um algoritmo de aproximação 2 para emparelhamento de máximo.

c. Mostre que o tamanho de um emparelhamento máximo em G é um limite inferior para o tamanho de qualquer cobertura de vértice para G.

d. Considere um emparelhamento maximal M em $G = (V, E)$. Seja $T = \{v \in V$: alguma aresta em M é incidente em $v\}$.

O que você pode dizer sobre o subgrafo de G induzido pelos vértices de G que não estão em T?

e. Conclua, pela parte (d) que $2|M|$ é o tamanho de uma cobertura de vértices para G.

f. Usando as partes (c) e (e), prove que o algoritmo guloso da parte (b) é um algoritmo de aproximação 2 para emparelhamento máximo.

35-5 Escalonamento de máquinas paralelas

No ***problema de escalonamento de máquinas paralelas***, temos n tarefas, $J_1, J_2,..., J_n$, onde cada tarefa J_k tem um tempo de processamento não negativo associado p_k. Também temos m máquinas idênticas, $M_1, M_2,...,M_m$. Qualquer tarefa pode ser executada em qualquer máquina. Um ***escalonamento*** especifica, para cada tarefa J_k, a máquina na qual ela é executada e o período de tempo durante o qual ela é executada. Cada tarefa J_k deve ser executada em alguma máquina M_i durante p_k unidades de tempo consecutivas e, durante esse período de tempo, nenhuma outra tarefa pode ser executada em M_i. Seja C_k o ***tempo de conclusão*** da tarefa J_k, isto é, o tempo em que a tarefa J_k conclui o processamento. Dado um escalonamento, definimos $C_{max} = \max_{1 \leq j \leq n} C_j$ como a ***duração*** da escalonamento. A meta é determinar um escalonamento cuja duração seja mínima.

Por exemplo, suponha que tenhamos duas máquinas M_1 e M_2 e quatro tarefas J_1, J_2, J_3, J_4, com $p_1 = 2$, $p_2 = 12$, $p_3 = 4$ e $p_4 = 5$. Então uma escalonamento possível executa, na máquina M_1, a tarefa J_1 seguida pela tarefa J_2 e, na máquina M_2, ela executa a tarefa J_4 seguido pela tarefa J_3. Para esse escalonamento, $C_1 = 2$, $C_2 = 14$, $C_3 = 9$, $C_4 = 5$ e $C_{max} = 14$. Um escalonamento ótimo executa J_2 na máquina M_1, e as tarefas J_1, J_3 e J_4 na máquina M_2. Para esse escalonamento, $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, $C_4 = 11$ e $C_{max} = 12$.

Dado um problema de escalonamento de máquinas paralelas, seja C^*_{max} a duração de um escalonamento ótimo.

a. Mostre que a duração ótima é no mínimo tão grande quanto o maior tempo de processamento, isto é,

$$C^*_{max} \geq \max_{1 \leq k \leq n} p_k .$$

b. Mostre que a duração ótima é no mínimo tão grande quanto a carga média da máquina, isto é,

$$C^*_{max} \geq \frac{1}{m} \sum_{1 \leq k \leq n} p_k .$$

Suponha que usamos o seguinte algoritmo guloso para escalonamento de máquinas paralelas: sempre que uma máquina estiver ociosa, programar qualquer tarefa que ainda não tenha sido escalonada.

c. Escreva pseudocódigo para implementar esse algoritmo guloso. Qual é o tempo de execução de seu algoritmo?

d. Para a programação retornada pelo algoritmo guloso, mostre que

$$C_{\max} \leq \frac{1}{m} \sum_{1 \leq k \leq n} p_k + \max_{1 \leq k \leq n} p_k$$

Conclua que esse algoritmo é um algoritmo polinomial de 2–aproximação.

35-6 Aproximação para uma árvore geradora máxima

Seja G = (V, E) um grafo não dirigido com pesos de arestas distintos $w(u, v)$ em cada aresta $(u, v) \in E$. Para cada vértice $v \in V$, seja $\max(v) = \max_{(u, v) \in E} \{w(u, v)\}$ a aresta de peso máximo incidente nesse vértice. Seja $S_G = \{\max(v) : v \in V\}$ o conjunto de arestas de pesos máximos incidentes em cada vértice, e seja T_G a árvore geradora de peso máximo de G, isto é, a árvore geradora de peso total máximo. Para qualquer subconjunto de arestas $E' \subseteq E$, defina $w(E') = \sum_{(u, v) \in E'} w(u, v)$.

a. Dê um exemplo de grafo com no mínimo quatro vértices para o qual $S_G = T_G$.

b. Dê um exemplo de grafo com no mínimo quatro vértices para o qual $S_G \neq T_G$.

c. Prove que $S_G \subseteq T_G$ para qualquer grafo G.

d. Prove que $w(T_G) \geq w(S_G)/2$ para qualquer grafo G.

e. Dê um algoritmo de tempo $O(V + E)$ para calcular uma 2-aproximação para a árvore geradora máxima.

35-7 Um algoritmo de aproximação para o problema da mochila 0-1

Lembre-se do problema da mochila na Seção 16.2. Há n itens, onde o i–ésimo item vale v_i dólares e pesa w_i libras. Temos também uma mochila que pode acondicionar no máximo W libras. Aqui, adicionamos as premissas de que cada peso w_i é no máximo W e que os itens estão indexados em ordem monotonicamente decrescente de seus valores: $v_1 \geq v_2 \geq \dots \geq v_n$.

No problema da mochila 0-1, desejamos determinar um subconjunto de itens cujo peso total seja no máximo W e cujo valor total seja máximo. O problema da mochila fracionário é como o problema da mochila 0-1, exceto que é permitido tomar uma fração de cada item, em vez de estarmos restritos a tomar um item inteiro ou nenhuma fração desse item. Se tomarmos uma fração x_i do item i, onde $0 \leq x_i \leq 1$, contribuímos com $x_i w_i$ para o peso da mochila e recebemos o valor $x_i v_i$. Nossa meta é desenvolver um algoritmo de aproximação 2 de tempo polinomial para o problema da mochila 0-1.

Para projetar um algoritmo de tempo polinomial, consideramos instâncias restringidas do problema da mochila 0-1. Dada uma instância I do problema da mochila, formamos instâncias restringidas I_j, para $j = 1, 2, \dots, n$, eliminando itens $1, 2, \dots, j - 1$ e exigindo que a solução inclua o item j (o item j inteiro, tanto no problema da mochila fracionário quanto no problema da mochila 0-1). Nenhum item é eliminado na instância I_1. Para a instância I_j, seja P_j uma solução ótima para o problema 0-1 e Q_j uma solução ótima para o problema fracionário.

a. Demonstre que uma solução ótima para a instância I do problema da mochila 0-1 seja pertença a $\{P_1, P_2, \dots ;P_n\}$.

b. Prove que podemos determinar uma solução ótima Q_j para o problema fracionário para a instância I_j incluindo o item j e depois usando o algoritmo guloso no qual, a cada etapa, tomamos o máximo possível do item k não escolhido no conjunto $\{j + 1, j + 2, \ldots, n\}$ com o valor máximo por libra v_i / w_i.

c. Prove que sempre podemos construir uma solução ótima Q_j para o problema fracionário para a instância I_j que inclua no máximo um item fracionário. Isto é, para todos os itens, exceto possivelmente um, incluímos na mochila o item inteiro ou nenhuma fração dele.

d. Dada uma solução ótima Q_j para o problema fracionário para a instância I_j, forme a solução R_j de Q_j eliminando quaisquer itens fracionários de Q_j. Seja $v(S)$ o valor total de itens tomados em uma solução S. Prove que $v(R_j) \geq v(Q_j)/2 \geq v(P_j)/2$.

e. Dê um algoritmo de tempo polinomial que retorne uma solução de valor máximo do conjunto $\{R_1, R_2, \ldots, R_n\}$ e prove que o seu algoritmo é um algoritmo polinomial de 2–aproximação para o problema da mochila 0-1.

NOTAS DO CAPÍTULO

Embora métodos que não calculam necessariamente soluções exatas sejam conhecidos há milhares de anos (por exemplo, métodos para aproximar o valor de) , a noção de um algoritmo de aproximação é muito mais recente. Hochbaum [172], credita a Garey, Graham e Ullman [128] e a Johnson [190] a formalização do conceito de um algoritmo de aproximação de tempo polinomial. O primeiro algoritmo desse tipo é frequentemente creditado a Graham [149].

Desde essas primeiras obras, milhares de algoritmos de aproximação têm sido projetados para ampla faixa de problemas e há profusão de literatura nessa área. Textos recentes de Ausiello *et al*. [26], Hochbaum [172] e Vazirani [345] tratam exclusivamente de algoritmos de aproximação, assim como os levantamentos de Shmoys [315] e de Klein e Young [207]. Vários outros textos, como os de Garey e Johnson [129] e de Papadimitriou e Steiglitz [271] também apresentam uma cobertura significativa de algoritmos de aproximação. Lawler, Lenstra, Rinnooy Kan e Shmoys [225] dão um tratamento extensivo de algoritmos de aproximação para o problema do caixeiro-viajante.

Papadimitriou e Steiglitz atribuem o algoritmo Approx-Vertex-Cover a F. Gavril e M. Yannakakis. O problema de cobertura de vértices foi estudado extensivamente (Hochbaum [172] apresenta uma lista com 16 algoritmos de aproximação diferentes para esse problema), mas todas as razões de aproximação são no mínimo $2 - o(1)$.

O algoritmo Approx-TSP-Tour aparece em um artigo por Rosenkrantz, Stearns e Lewis [298]. Christofides melhorou esse algoritmo e apresentou um algoritmo de aproximação 3/2 para o problema do caixeiro-viajante com a desigualdade triangular. Arora [22] e Mitchell [257] mostraram que, se os pontos estão sobre um plano euclidiano, existe um esquema de aproximação de tempo polinomial. O Teorema (35.3) se deve a Sahni e Gonzalez [301].

A análise da heurística gulosa para o problema de cobertura de conjuntos é modelada com base na prova publicada por Chvátal [68] de um resultado mais geral; o resultado básico como apresentado aqui se deve a Johnson [190] e Lovász [238].

O algoritmo Approx-Subset-Sum e sua análise são modelados livremente conforme os algoritmos de aproximação relacionados para os problemas da mochila e de soma de subconjuntos por Ibarra e Kim [187].

O Problema 35-7 é uma versão combinatória de um resultado mais geral para aproximação de programas do tipo do programa da mochila com inteiros por Bienstock e McClosky [45].

O algoritmo aleatorizado para satisfazibilidade MAX-3-CNF está implícito no trabalho de Johnson [190]. O algoritmo de cobertura de vértices ponderada é de Hochbaum [171]. A Seção 35.4 apenas toca no poder da aleatorização e da programação linear para o projeto de algoritmos de aproximação. Uma combinação dessas duas ideias gera uma técnica denominada “arredondamento aleatorizado”, que formula um problema como um programa

linear inteiro, resolve o relaxamento linear e interpreta as variáveis na solução como probabilidades. Então, essas probabilidades são usadas para ajudar a orientar a solução do problema original. Essa técnica foi usada inicialmente por Raghavan e Thompson [290], e teve muitos usos subsequentes. (Consulte o levantamento apresentado por Motwani, Naor e Raghavan [261]. Entre várias outras ideias notáveis recentes na área de algoritmos de aproximação citamos o método dual primitivo (consulte o levantamento apresentado por Goemans e Williamson [135]), a determinação de cortes esparsos para uso em algoritmos de divisão e conquista [229] e o uso de programação semidefinida [134].

Como mencionamos nas notas do Capítulo 34, resultados recentes em provas que podem ser verificadas probabilisticamente resultaram em limites mais baixos para a capacidade de aproximação de muitos problemas, incluídos vários que aparecem neste capítulo. Além das referências dadas ali, o capítulo por Arora e Lund [23] contém uma boa descrição da relação entre provas que podem ser verificadas probabilisticamente e a dificuldade de aproximar diversos problemas.

---

[1] Quando a razão de aproximação é independente de n, usamos os termos "razão de aproximação de r" e "algoritmo de aproximação r", que indica a não dependência de n.

Parte

APÊNDICE: FUNDAMENTOS DE MATEMÁTICA

INTRODUÇÃO

Quando analisamos algoritmos, muitas vezes, precisamos recorrer a um conjunto de ferramentas matemáticas. Algumas dessas ferramentas são tão simples quanto a álgebra que aprendemos no segundo grau, mas outras podem ser novas para você. Na Parte I, vimos como tratar notações assintóticas e resolver recorrências. Este apêndice compreende um compêndio de vários outros conceitos e métodos que empregamos para analisar algoritmos. Como observamos na introdução à Parte I, é possível que você já tenha visto grande parte do material neste apêndice antes de ler este livro (embora ocasionalmente as convenções específicas de notação que utilizamos possam ser diferentes das que viu em outros lugares). Consequentemente, você deve tratar este apêndice como material de referência. No entanto, como no restante deste livro, incluímos exercícios e problemas para que você possa melhorar seus conhecimentos nessas áreas.

O Apêndice A oferece métodos para avaliar e limitar somatórias, que ocorrem frequentemente na análise de algoritmos. Muitas das fórmulas apresentadas aqui aparecem em qualquer livro didático de cálculo, mas você verá que é conveniente ter uma compilação desses métodos em um único lugar.

O Apêndice B contém definições e notações básicas para conjuntos, relações, funções, grafos e árvores, e também apresenta algumas propriedades básicas desses objetos matemáticos.

O Apêndice C começa com princípios elementares de contagem: permutações, combinações e semelhantes. O restante contém definições e propriedades de probabilidade básica. Grande parte dos algoritmos deste livro não exige nenhum conhecimento de probabilidade para sua análise e, assim, você poderá omitir facilmente as últimas seções do capítulo em uma primeira leitura, até mesmo sem folheá-las. Mais tarde, quando encontrar uma análise probabilística que queira entender melhor, verá que o Apêndice C é bem organizado como material de referência.

O apêndice D define matrizes, operações com matrizes e algumas de suas propriedades básicas. É provável que você já tenha estudado a maioria desse material se já fez algum curso de álgebra linear, mas verá que é útil ter um lugar separado para procurar nossas notações e definições.

A SOMATÓRIOS

Quando um algoritmo contém um constructo de controle iterativo como um laço **while** ou **for**, podemos expressar seu tempo de execução como a soma dos tempos gastos em cada execução do corpo do laço. Por exemplo, vimos na Seção 2.2 que a j-ésima iteração da ordenação por inserção demorou um tempo proporcional a j no pior caso. Somando o tempo gasto em cada iteração, obtivemos o somatório (ou a série)

$$\sum_{j=2}^{n} j$$

Quando avaliamos esse somatório, obtivemos um limite de Q(n_2) para o tempo de execução do pior caso do algoritmo. Esse exemplo ilustra por que você deve saber como lidar com somatórias e limitá-las.

A Seção A.1 apresenta uma lista de várias fórmulas básicas que envolvem somatórios. A Seção A.2 oferece técnicas úteis para limitar somatórias. Apresentamos as fórmulas da Seção A.1 sem provas, embora provas para algumas delas apareçam na Seção A.2 para ilustrar os métodos dessa seção. Você pode encontrar a maioria das outras provas em qualquer livro didático de cálculo.

A.1 FÓRMULAS E PROPRIEDADES DE SOMATÓRIOS

Dada uma sequência de números a_1, a_2,..., a_n, podemos escrever a soma finita $a_1 + a_2 + \ldots + a_n$, onde n é um inteiro não negativo, como

$$\sum_{k=1}^{n} a_k .$$

Se $n = 0$, o valor do somatório é definido como 0. O valor de uma série finita é sempre bem definido, e seus termos podem ser somados em qualquer ordem.

Dada uma sequência infinita de números a_1, a_2,..., podemos escrever a soma infinita $a_1 + a_2 + \ldots$ como

$$\sum_{k=1}^{\infty} a_k,$$

cujo significado interpretamos como

$$\lim_{x \to \infty} \sum_{k=1}^{n} a_k.$$

Se o limite não existe, a série ***diverge***; caso contrário, ela ***converge***. Os termos de uma série convergente nem sempre podem ser somados em qualquer ordem. Contudo, podemos reorganizar os termos de uma ***série absolutamente convergente***, isto é, uma série $\sum_{k=1}^{\infty} a_k$ para a qual a série $\sum_{k=1}^{\infty} |a_k|$ também converge.

Linearidade

Para qualquer número real c e quaisquer sequências finitas $a_1, a_2, \ldots, a_n$ e $b_1, b_2, \ldots, b_n$,

$$\sum_{k=1}^{n} (ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k.$$

A propriedade de linearidade também se aplica a séries convergentes infinitas.

Podemos explorar a propriedade de linearidade para lidar com somatórias que incorporam notação assintótica. Por exemplo,

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right).$$

Nessa equação, a notação Q no lado esquerdo se aplica à variável k mas, no lado direito, ela se aplica a n. Também podemos aplicar tais manipulações a séries convergentes infinitas.

Série aritmética

O somatório

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n,$$

é uma ***série aritmética*** e tem o valor

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1) \quad \text{(A.1)}$$

$$= \Theta(n^2) \quad \text{(A.2)}$$

Somas de quadrados e cubos

Temos os seguintes somatórios de quadrados e cubos:

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}, \qquad \text{(A.3)}$$

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}. \qquad \text{(A.4)}$$

Série geométrica

Para números reais $x \neq 1$, o somatório

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

é uma ***série geométrica*** ou ***exponencial*** e tem o valor

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}. \qquad \text{(A.5)}$$

Quando o somatório é infinito e $|x| < 1$, temos a série geométrica decrescente infinita

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}. \qquad \text{(A.6)}$$

Como adotamos $0^0 = 1$, essas fórmulas valem também quando $x = 0$.

Série harmônica

Para inteiros positivos n, o n-ésimo ***número harmônico*** é

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^{n} \frac{1}{k} \\ &= \ln n + o(1). \end{aligned} \qquad \text{(A.7)}$$

(Provaremos um limite relacionado na Seção A.2.)

Integração e diferenciação de séries

Integrando e diferenciando as fórmulas citadas, surgem fórmulas adicionais. Por exemplo, diferenciando ambos os lados da série geométrica infinita (A.6) e multiplicando por x, obtemos

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1 - x)^2} \qquad \text{(A.8)}$$

para $|x| < 1$.

Séries telescópicas

Para qualquer sequência $a_1, a_2, \ldots, a_n$,

$$\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0, \tag{A.9}$$

já que cada um dos termos $a_1, a_2,..., a_{n-1}$ é somado exatamente uma vez e subtraído exatamente uma vez. Dizemos que a soma é **telescópica**. De modo semelhante,

$$\sum_{k=1}^{n-1}(a_k - a_{k+1}) = a_n - a_0,$$

Como exemplo de uma soma telescópica, considere a série

$$\sum_{k=1}^{n-1}\frac{1}{k(k+1)}.$$

Visto que podemos reescrever cada termo como

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{1+k},$$

obtemos

$$\begin{aligned}\sum_{k=1}^{n-1}\frac{1}{k(k+1)} &= \sum_{k=1}^{n-1}\left(\frac{1}{k} - \frac{1}{k+1}\right)\\ &= 1 - \frac{1}{n}\end{aligned}$$

Produtos

Podemos escrever o produto finito $a_1 a_2 \ldots a_n$ como

$$\prod_{k=1}^{n} a_{k.}$$

Se $n = 0$, o valor do produto é definido como 1. Podemos converter uma fórmula com um produto em uma fórmula com um somatório utilizando a identidade

$$\lg\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \lg a_k.$$

Exercícios

A.1-1 Determine uma fórmula simples para $\sum_{k=1}^{n}(2k-1)$.

A.1-2 ★
Mostre que $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$, manipulando a série harmônica.

A.1-3 Mostre que $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$ para $|x| < 1$.

A.1-4 ★
Mostre que $\sum_{k=0}^{\infty} (k-1)/2^k = 0$.

A.1-5 ★
Avalie a soma $\sum_{k=1}^{\infty} (2k+1)x^{2k}$ para $|x| < 1$.

A.1-6 Prove que $\sum_{k=1}^{n} O(f_k(i)) = O(\sum_{k=1}^{n} (f_k(i))$ usando a propriedade de linearidade de somatórios.

A.1-7 Avalie o produto $\prod_{k=1}^{n} 2 \cdot 4^k$.

A.1-8 ★
Avalie o produto $\prod_{k=2}^{n} (1 - 1/k^2)$.

A.2 LIMITANDO SOMATÓRIOS

Temos muitas técnicas disponíveis para limitar os somatórios que descrevem os tempos de execução de algoritmos. Aqui apresentamos alguns dos métodos mais frequentemente empregados.

Indução

O modo mais comum para avaliar uma série é usar indução. Como exemplo, vamos provar que a série aritmética $\sum_{k=1}^{n} k$ *k* tem o valor 1 *n* (*n* + 1). É fácil verificar essa afirmativa para $n = 1$. Admitimos indutivamente que ela seja válida para *n* e provamos que ela vale para $n + 1$. Temos

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1)$$

$$= \frac{1}{2} n(n+1) + (n+1)$$

$$= \frac{1}{2}(n+1)(n+2)$$

Nem sempre é necessário adivinhar o valor exato de um somatório para usar indução. Em vez disso, podemos usar indução para provar um limite para um somatório. Como exemplo, vamos provar que a série geométrica $\sum_{k=0}^{n} 3^k$ é $O(3_n)$. Mais especificamente, vamos provar que $\sum_{k=0}^{n} 3^k \leq c3_n$ para alguma constante *c*. Para a condição inicial $n = 0$, temos $\sum_{k=0}^{n} 3^k = 1 \cdot c$ desde que $c \geq 1$. Supondo que o limite vale para *n*, vamos provar que ele é válido para $n + 1$. Temos

$$
\begin{aligned}
\sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^{n} 3^k + 3^{n+1} \\
&\le c3^n + 3^{n+1} \qquad \text{(por hipótese de indução)} \\
&= \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1} \\
&\le c3^{n+1}
\end{aligned}
$$

desde que $(1/3 + 1/c) \le 1$ ou, o que é equivalente, $c \ge 3/2$. Assim, $\sum_{k=0}^{n} 3^k = O(3_n)$, como queríamos demonstrar.

Temos de tomar cuidado quando usarmos notação assintótica para provar limites por indução. Considere a seguinte prova falaciosa que $\sum_{k=1}^{n} k = O(n)$. Certamente, $\sum_{k=1}^{n} k = O(1)$. Supondo que o limite é válido para n, agora o provamos para $n + 1$:

$$
\begin{aligned}
\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1) \\
&= O(n) + (n+1) \qquad \Leftarrow \textit{errado}!! \\
&= O(n+1)
\end{aligned}
$$

O erro no argumento é que a "constante" oculta pelo "*O* grande" cresce com n e, portanto, não é constante. Não mostramos que a mesma constante funciona para *todo* n.

Limitando termos

Às vezes, podemos obter um bom limite superior para uma série limitando cada termo da série, e muitas vezes basta utilizar o maior termo para limitar os outros. Por exemplo, um limite superior rápido para a série aritmética (A.1) é

$$
\begin{aligned}
\sum_{k=1}^{n} k &\le \sum_{k=1}^{n} n \\
&= n^2 .
\end{aligned}
$$

Em geral, para uma série $\sum_{k=1}^{n} a_k$, se fizermos $a_{\max} = \max{}_{1 k n} a_k$, então

$$
\sum_{k=1}^{n} a_k \le n \cdot a_{\text{máx}} .
$$

A técnica de limitar cada termo em uma série pelo maior termo é um método fraco quando a série pode ser de fato limitada por uma série geométrica. Dada a série $\sum_{k=0}^{n} a_k$, suponha que $a_{k+1} / a_k \le r$ para todo $k \ge 0$, onde $0 < r < 1$ é uma constante. Podemos limitar a soma por uma série geométrica decrescente infinita, já que $a_k \le a_0 r^k$ e, assim,

$$
\begin{aligned}
\sum_{k=0}^{n} a_k &\le \sum_{k=0}^{\infty} a_0 r^k \\
&= a_0 \sum_{k=0}^{\infty} r^k \\
&= a_0 \frac{1}{1-r} .
\end{aligned}
$$

Podemos aplicar esse método para limitar o somatório $\sum_{k=1}^{n} (k/3^k)$. Para iniciar o somatório em $k = 0$, nós o reescrevemos como $\sum_{k=1}^{n} ((k + 1)/3^{k+1})$. O primeiro termo ($a_0$) é 1/3, e a razão ($r$) entre os termos sucessivos é

$$\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} = \frac{1}{3} \cdot \frac{k+2}{k+1} \\ \leq \frac{2}{3}$$

para todo $k \geq 0$. Assim, temos

$$\sum_{k=1}^{\infty} \frac{k}{3^k} = \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}} \\ \leq \frac{1}{3} \cdot \frac{1}{1-2/3} \\ = 1.$$

Um erro comum na aplicação desse método é mostrar que a razão entre termos sucessivos é menor que 1 e depois decidir que o somatório é limitado por uma série geométrica. Um exemplo é a série harmônica infinita, que diverge, já que

$$\sum_{k=1}^{\infty} \frac{1}{k} = \lim_{x \to \infty} \sum_{k=1}^{n} \frac{1}{k} \\ = \lim_{x \to \infty} \Theta(\lg n) \\ = \infty.$$

A razão entre o $(k + 1)$-ésimo termo e o k-ésimo termo nessa série é $k/(k + 1) < 1$, mas a série não é limitada por uma série geométrica decrescente. Para limitar uma série por uma série geométrica, devemos mostrar que existe um $r < 1$ que é uma *constante*, tal que a razão entre todos os pares de termos sucessivos nunca exceda r. Na série harmônica, não existe tal r porque a razão se torna arbitrariamente próxima de 1.

Quebra de somatórios

Um dos modos de obter limites para um somatório difícil é expressar a série como a soma de duas ou mais séries particionando a faixa do índice e depois limitando cada uma das séries resultantes. Por exemplo, suponha que tentamos determinar um limite inferior para a série aritmética $\sum_{k=1}^{n} k$, que, como já vimos, tem um limite superior de n_2. Poderíamos tentar limitar cada termo no somatório pelo menor termo mas, como esse termo é 1, obtemos um limite inferior de n para o somatório — bem longe do nosso limite superior de n_2.

Podemos obter um limite inferior melhor quebrando primeiro o somatório. Por conveniência, suponha que n seja par. Temos

$$\begin{aligned}\sum_{k=1}^{n} k &= \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \\ &\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} (n/2) \\ &= (n/2)^2 \\ &= \Omega(n^2),\end{aligned}$$

que é um limite assintoticamente justo, visto que $\sum_{k=1}^{n} k = O(n^2)$.

Quando se trata de um somatório que surge da análise de um algoritmo, muitas vezes podemos quebrar o somatório e ignorar um número constante de termos iniciais. Em geral, essa técnica se aplica quando cada termo a_k em um somatório $\sum_{k=0}^{n} a_k$ é independente de n. Então, para qualquer constante $k_0 > 0$, podemos escrever

$$\begin{aligned}\sum_{k=0}^{n} a_k &= \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k \\ &= \Theta(1) + \sum_{k=k_0}^{n} a_k\end{aligned}$$

já que os termos iniciais do somatório são constantes e há um número constante deles. Então, podemos usar outros métodos para limitar $\sum_{k=0}^{n} 3^k a_k$. Por exemplo, para determinar um limite superior assintótico para

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k} ,$$

observamos que a razão entre termos consecutivos é

$$\begin{aligned}\frac{(k+1)^2 / 2^{k+1}}{k^2 / 2^k} &= \frac{(k+1)^2}{2k^2} \\ &\leq \frac{8}{9}\end{aligned}$$

se $k \geq 3$. Assim, o somatório também pode ser quebrado em

$$\begin{aligned}\sum_{k=0}^{\infty} \frac{k^2}{2^k} &= \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \\ &\leq \sum_{k=0}^{2} \frac{k^2}{2^k} + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k \\ &= O(1),\end{aligned}$$

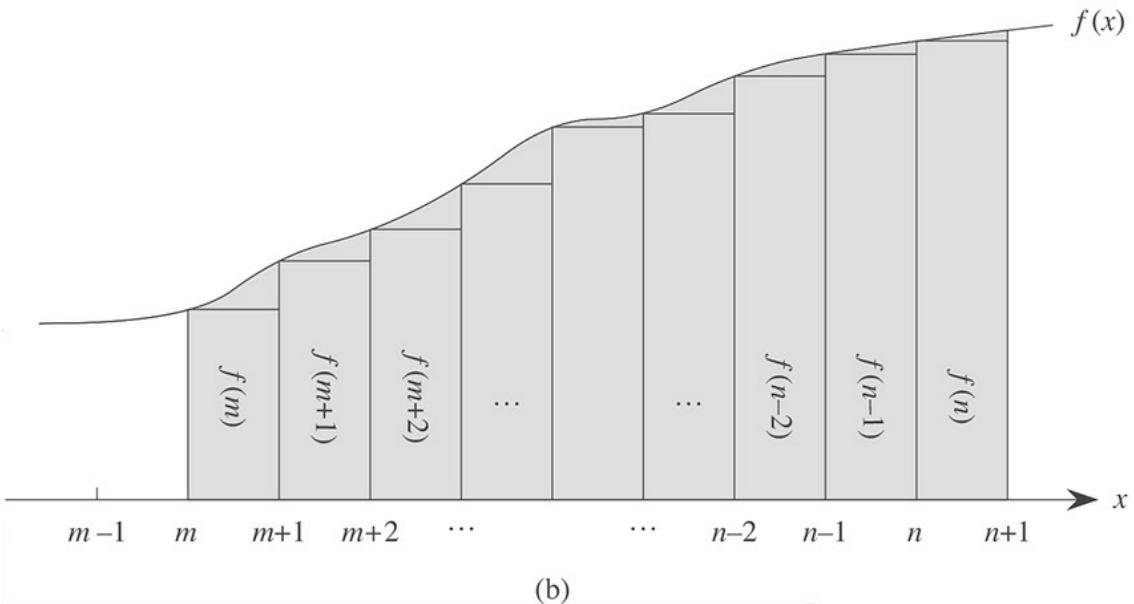
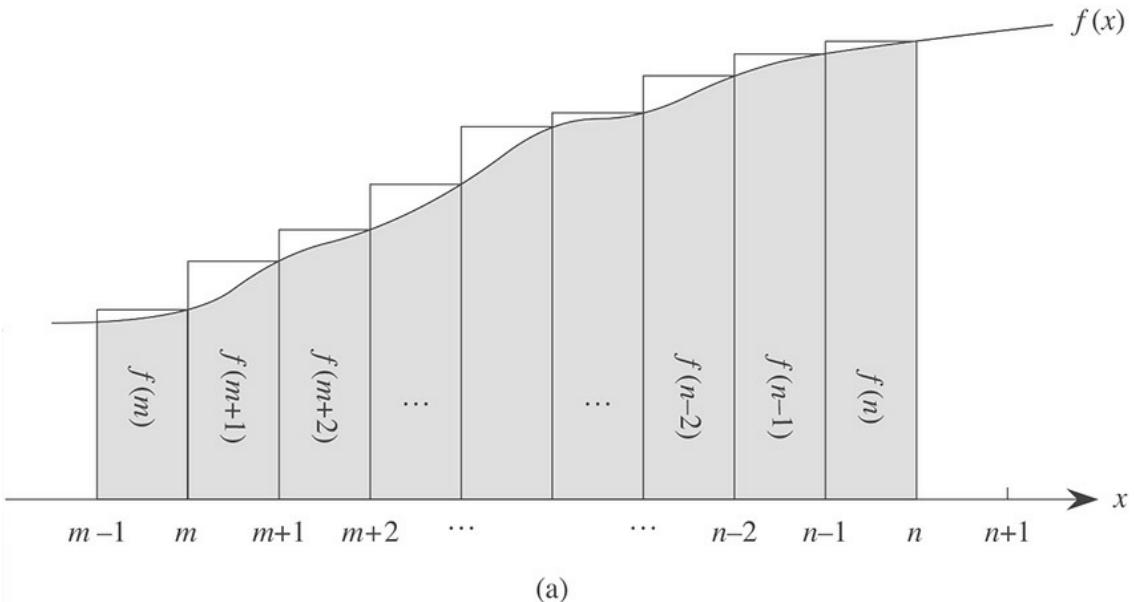
já que o primeiro somatório tem um número constante de termos e o segundo somatório é uma série geométrica decrescente.

A técnica de quebrar somatórios pode nos ajudar a determinar limites assintóticos em situações muito mais difíceis. Por exemplo, podemos obter um limite de $O(\lg n)$ para a série harmônica (A.7):

$$H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

Fazemos isso dividindo a faixa 1 a n em $\lg n + 1$ pedaços e limitando por 1 a contribuição de cada pedaço. Para $i = 0, 1, \ldots, \lg n$, o i-ésimo pedaço consiste nos termos que começam em $1/2^i$ e vão até $1/2^{i+1}$, mas não o incluem. O último pedaço pode conter termos que não estão na série original e, assim, temos

$$\begin{aligned}\sum_{k=1}^{n} \frac{1}{k} &\le \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \\ &\le \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} \\ &= \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \\ &= \lg n + 1. \end{aligned} \tag{A.10}$$

Aproximação por integrais

Quando um somatório tem a forma $\sum_{k=m}^{n} f(k)$, onde $f(k)$ é uma função monotonicamente crescente, podemos aproximá-lo por integrais:

$$\int_{m-1}^{n} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)\,dx. \tag{A.11}$$

A Figura A.1 justifica essa aproximação. O somatório é representado como a área dos retângulos na figura, e a integral é a região sombreada sob a curva. Quando $f(k)$ é uma função monotonicamente decrescente, podemos usar um método semelhante para determinar os limites

$$\int_{m}^{n+1} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)\,dx. \tag{A.12}$$

Figura A.1 Aproximação de $\sum_{k=m}^{n} f(k)$ por integrais. A área de cada retângulo é mostrada dentro do retângulo, e a área total dos retângulos representa o valor do somatório. A integral é representada pela área sombreada sob a curva. Comparando as áreas em **(a)**, obtemos $\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(k)$, e depois, deslocando os retângulos uma unidade para a direita, obtemos $\sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)dx$ em **(b)**.

A aproximação por integral (A.12) nos dá uma estimativa restrita para o n-ésimo número harmônico. Para um limite inferior, obtemos

$$\begin{aligned}\sum_{k=1}^{n} \frac{1}{k} &\ge \int_{1}^{n+1} \frac{dx}{x} \\ &= \ln(n+1).\end{aligned} \tag{A.13}$$

Para o limite superior, derivamos a desigualdade

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n} \frac{dx}{x}$$
$$= \ln n,$$

que produz o limite

$$\sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1 \tag{A.14}$$

Exercícios

A.2-1 Mostre que $\sum_{k=1}^{n} 1/k^2$ é limitado por cima por uma constante.

A.2-2 Determine um limite superior assintótico para o somatório

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil.$$

A.2-3 Mostre que o n-ésimo número harmônico é $\Omega(\lg n)$, dividindo o somatório.

A.2-4 Aproxime $\sum_{k=1}^{n} k^3$ com uma integral.

A.2-5 Por que não usamos a aproximação por integral (A.12) diretamente em $\sum_{k=1}^{n} 1/k$ para obter um limite superior para o n-ésimo número harmônico?

Problemas

A-1 Limitando somatórios

Dê limites assintoticamente justos para os somatórios a seguir. Suponha que $r \geq 0$ e $s \geq 0$ são constantes.

a. $\sum_{k=1}^{n} k^r$.

b. $\sum_{k=1}^{n} \lg^s k$.

c. $\sum_{k=1}^{n} k^r \lg^s k$.

Notas do apêndice

Knuth [209] é uma excelente referência para o material apresentado aqui. O leitor pode encontrar propriedades básicas de séries em qualquer bom livro de cálculo, como Apostol [18] ou Thomas *et al.* [334].

B CONJUNTOS ETC.

Muitos capítulos deste livro mencionam elementos da matemática discreta. Este apêndice reexamina de forma mais completa as notações, definições e propriedades elementares de conjuntos, relações, funções, grafos e árvores. Para os leitores que já estão bem versados nesses assuntos, basta dar uma olhada rápida neste capítulo.

B.1 CONJUNTOS

Um ***conjunto*** é uma coleção de objetos distintos, denominados ***elementos*** ou ***membros*** do conjunto. Se um objeto x é um elemento de um conjunto S, escrevemos $x \in S$ (lê-se "x é um membro de S", "x é um elemento de S", "x pertence a S" ou, de modo mais abreviado, "x está em S"). Se x não é um elemento de S, escrevemos que $x \notin S$. Podemos descrever um conjunto relacionando explicitamente seus elementos como uma lista entre chaves. Por exemplo, podemos definir um conjunto S contendo exatamente os números 1, 2 e 3, escrevendo $S = \{1, 2, 3\}$. Como 2 é um elemento do conjunto S, podemos escrever $2 \in S$; como 4 não é um elemento do conjunto, temos $4 \notin S$. Um conjunto não pode conter o mesmo objeto mais de uma vez,[1] e seus elementos não são ordenados. Dois conjuntos A e B são ***iguais***, expresso por $A = B$, se eles contêm os mesmos elementos. Por exemplo, $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$.

Adotamos notações especiais para conjuntos encontrados com frequência:

- /0 denota o ***conjunto vazio***, isto é, o conjunto que não contém nenhum elemento.
- denota o conjunto de ***números inteiros***, isto é, o conjunto $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- denota o conjunto de ***números reais***.
- denota o conjunto de ***números naturais***, isto é, o conjunto $\{0, 1, 2, \ldots\}$.[2]

Se todos os elementos de um conjunto A estão contidos em um conjunto B, isto é, se $x \in A$ implica $x \in B$, escrevemos $A \subseteq B$ e dizemos que A é um ***subconjunto*** de B. Um conjunto A é um ***subconjunto próprio*** de B, representado por $A \subset B$, se $A \subseteq B$, mas $A \neq B$. (Alguns autores usam o símbolo "$\subset$" para denotar a relação de subconjunto, em vez da relação de subconjunto próprio.) Para qualquer conjunto A, temos $A \subseteq A$. No caso de dois conjuntos A e B, temos $A = B$ se e somente se $A \subseteq B$ e $B \subseteq A$. Para três conjuntos A, B e C quaisquer, se $A \subseteq B$ e $B \subseteq C$, então $A \subseteq C$. Para qualquer conjunto A, temos 0/ $\subseteq A$.

Algumas vezes, definimos conjuntos em termos de outros conjuntos. Dado um conjunto A, podemos definir um conjunto $B \subseteq A$ enunciando uma propriedade que distingue os elementos de B. Por exemplo, podemos definir o conjunto dos números inteiros pares por $\{x : x \in$ e $x/2$ é um inteiro$\}$. Nessa notação, o sinal de dois-pontos significa "tal que". (Alguns autores usam uma barra vertical em vez do sinal de dois-pontos.)

Dados dois conjuntos A e B, também podemos definir novos conjuntos aplicando ***operações de conjuntos***:

- A ***interseção*** de conjuntos A e B é o conjunto

$$A \cap B = \{x : x \in A \text{ e } x \in B\} .$$

- A ***união*** de conjuntos A e B é o conjunto

  $A \cup B = \{x : x \in A \text{ ou } x \in B\}$.

- A ***diferença*** entre dois conjuntos A e B é o conjunto

  $A - B = \{x : x \in A \text{ e } x \notin B\}$.

  As operações de conjuntos obedecem às seguintes leis:

Leis do conjunto vazio:

$$A \cap \emptyset = \emptyset \ ,$$
$$A \cup \emptyset = A \ .$$

Leis de idempotência:

$$A \cap A = A \ ,$$
$$A \cup A = A \ .$$

Leis comutativas:

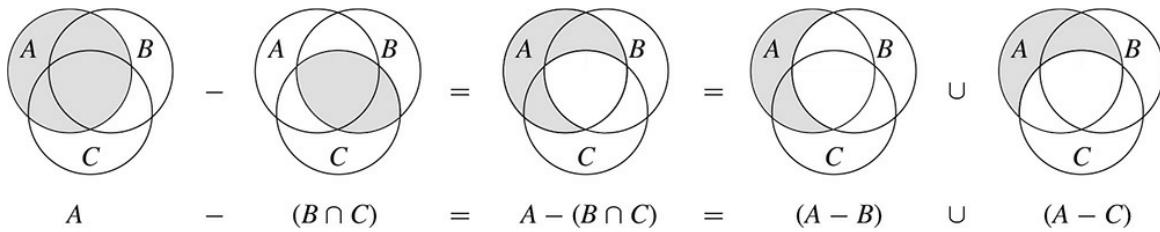
$$A \cap B = B \cap A \ ,$$
$$A \cup B = B \cup A \ .$$

Leis associativas:

$$A \cap (B \cap C) = (A \cap B) \cap C \ ,$$
$$A \cup (B \cup C) = (A \cup B) \cup C \ .$$

Leis distributivas:

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \ , \qquad \text{(B.1)}$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \ .$$

Leis da absorção:

$$A \cap (A \cup B) = A \ ,$$
$$A \cup (A \cap B) = A \ .$$

Leis de DeMorgan:

$$A - (B \cap C) = (A - B) \cup (A - C) \ ,$$
$$A - (B \cup C) = (A - B) \cap (A - C) \ . \qquad \text{(B.2)}$$

Figura B.1 Um diagrama de Venn que ilustra a primeira das leis de DeMorgan (B.2). Cada um dos conjuntos *A*, *B* e *C* é representado como um círculo.

A Figura B.1 ilustra a primeira das leis de DeMorgan por meio de um ***diagrama de Venn*****:** uma imagem gráfica na qual os conjuntos são representados como regiões do plano.

Muitas vezes, todos os conjuntos em consideração são subconjuntos de algum conjunto maior U denominado ***universo***. Por exemplo, se estivermos considerando vários conjuntos formados somente por inteiros, o conjunto de inteiros é um universo adequado. Dado um universo U, definimos o ***complemento*** de um conjunto A como $A = U - A = \{x : x \in U \text{ e } x \notin A\}$ Para qualquer conjunto $A \subseteq U$, temos as seguintes leis:

$$\overline{\overline{A}} = A \ ,$$
$$A \cap \overline{A} = \emptyset \ ,$$
$$A \cup \overline{A} = U \ .$$

Podemos reescrever as leis de DeMorgan (B.2) com complementos de conjuntos. Para dois conjuntos quaisquer $B, C \subseteq U$, temos

$$\overline{B \cap C} = \overline{B} \cup \overline{C} \ ,$$
$$\overline{B \cup C} = \overline{B} \cap \overline{C} \ .$$

Dois conjuntos A e B são ***disjuntos*** se não têm nenhum elemento em comum, isto é, se $A \cap B = /0$. Uma coleção $\mathcal{S} = \{S_i\}$ de conjuntos não vazios forma uma ***partição*** de um conjunto S se:

- os conjuntos são ***disjuntos aos pares***, isto é, $S_i, S_j \in \mathcal{S}$ e $i \neq j$ implicam $S_i \cap S_j = 0/$ e
- sua união é S, isto é,

$$S = \bigcup_{S_i \in \mathcal{S}} S_i .$$

Em outras palavras, $\mathcal{S}$ forma uma partição de S se cada elemento de S aparece em exatamente um $S_i \in \mathcal{S}$.

O número de elementos em um conjunto é a ***cardinalidade*** (ou ***tamanho***) do conjunto, denotada por $|S|$. Dois conjuntos têm a mesma cardinalidade se seus elementos podem ser colocados em uma correspondência de um para um. A cardinalidade do conjunto vazio é $|/0| = 0$. Se a cardinalidade de um conjunto é um número natural, dizemos que o conjunto é ***finito***; caso contrário, ele é ***infinito***. Um conjunto infinito que pode ser colocado em uma correspondência de um para um com os números naturais é ***infinito contável***; caso contrário, ele é ***não contável***. Os inteiros são contáveis, mas os reais são não contáveis.

Para quaisquer dois conjuntos finitos A e B, temos a identidade

$$|A \cup B| = |A| + |B| - |A \cap B| \ , \tag{B.3}$$

da qual podemos concluir que

$$|A \cup B| \leq |A| + |B| .$$

Se A e B são disjuntos, então $|A \cap B| = 0$ e, portanto, $|A \cup B| = |A| + |B|$. Se $A \subseteq B$, então $|A| \leq |B|$.

Um conjunto finito de n elementos às vezes é denominado ***n-conjunto***. Um conjunto de um elemento é denominado ***conjunto unitário.*** Um subconjunto de k elementos de um conjunto, às vezes, é denominado ***k-subconjunto***.

Denotamos o conjunto de todos os subconjuntos de um conjunto S, incluindo o conjunto vazio e o próprio conjunto S, por 2^S ; denominamos 2^S o ***conjunto potência*** de S. Por exemplo, $2^{\{a,b\}} = \{0/, \{a\}, \{b\}, \{a, b\}\}$. O conjunto potência de um conjunto finito S tem cardinalidade $2^{|S|}$ (veja o Exercício B.1-5).

Às vezes, utilizamos estruturas semelhantes a conjuntos nas quais os elementos estão ordenados. Um ***par ordenado*** de dois elementos a e b é denotado por (a, b) e definido formalmente como o conjunto $(a, b) = \{a, \{a, b\}\}$. Assim, o par ordenado (a, b) *não* é o mesmo que o par ordenado (b, a).

O ***produto cartesiano*** de dois conjuntos A e B, denotado por $A \times B$, é o conjunto de todos os pares ordenados tais que o primeiro elemento do par é um elemento de A e o segundo é um elemento de B. Em termos mais formais,

$$A \times B = \{(a, b) : a \in A \text{ e } b \in B\} .$$

Por exemplo, $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$. Quando A e B são conjuntos finitos, a cardinalidade de seu produto cartesiano é

$$|A \times B| = |A| \cdot |B| . \quad \text{(B.4)}$$

O produto cartesiano de n conjuntos $A_1, A_2, \ldots, A_n$ é o conjunto de ***n-tuplas***

$$A_1 \times A_2 \times \cdots \times A_n = (a_1, a_2, \ldots, a_n) : a_i \in A_i, i = 1, 2, \ldots, n) ,$$

cuja cardinalidade é

$$|A_1 \times A_2 \times \ldots \times A_n| = |A_1| \times |A_2| \ldots |A_n|$$

se todos os conjuntos são finitos. Denotamos um produto cartesiano de n termos em um único conjunto A pelo conjunto

$$A^n = A \times A \times \ldots \times A ,$$

cuja cardinalidade é $|A_n| = |A|^n$ se A é finito. Também podemos ver uma n-tupla como uma sequência finita de comprimento n (veja a seção B.3).

Exercícios

B.1-1 Esboçe diagramas de Venn que ilustrem a primeira das leis distributivas (B.1).

B.1-2 Prove a generalização das leis de DeMorgan para qualquer coleção finita de conjuntos:

$$\overline{A_1 \cap A_2 \cap \cdots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_n}$$

$$\overline{A_1 \cup A_2 \cup \cdots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}$$

B.1-3 ★

Prove a generalização da equação (B.3), que é denominada ***princípio de inclusão e exclusão***:

$$
\begin{aligned}
|A_1 \cup A_2 \cup \cdots \cup A_n| = \\
& |A_1| + |A_2| + \cdots + |A_n| \\
& -|A_1 \cap A_2| - |A_1 \cap A_3| - \cdots && \text{(todos os pares)} \\
& +|A_1 \cap A_2 \cap A_3| + \cdots && \text{(todas as triplas)} \\
& \quad \vdots \\
& +(-1)^{n-1} |A_1 \cap A_2 \cap A_3| \ .
\end{aligned}
$$

B.1-4 Mostre que o conjunto de números naturais ímpares é contável.

B.1-5 Mostre que, para qualquer conjunto finito S, o conjunto potência 2^S tem $2^{|S|}$ (isto é, existem $2^{|S|}$ subconjuntos distintos de S).

B.1-6 Dê uma definição indutiva para uma n-tupla estendendo a definição dada para um par ordenado.

B.2 RELAÇÕES

Uma ***relação binária*** R para dois conjuntos A e B é um subconjunto do produto cartesiano $A \times B$. Se $(a, b) \in R$, às vezes, escrevemos $a\ R\ b$. Quando dizemos que R é uma relação binária *em* um conjunto A, queremos dizer que R é um subconjunto de $A \times A$. Por exemplo, a relação "menor que" para os números naturais é o conjunto $\{(a, b) : a, b \in$ e $a < b\}$. Uma relação n-ária para os conjuntos $A_1, A_2, \ldots, A_n$ é um subconjunto de $A_1 \times A_2 \times \ldots \times A_n$.

Uma relação binária $R \subseteq A \times A$ é ***reflexiva*** se

$a\ R\ a$

para todo $a \in A$. Por exemplo, "=" e "≤" são relações reflexivas em , mas "<" não é. A relação R é ***simétrica*** se

$a\ R\ b$ implica $b\ R\ a$

para todo $a, b \in A$. Por exemplo, "=" é simétrica, mas "<" e "≤ " não são. A relação R é ***transitiva*** se

$a\ R\ b$ e $b\ R\ c$ implicam $a\ R\ c$

para todo $a, b, c \in A$. Por exemplo, as relações "<", "≤" e "=" são transitivas, mas a relação $R = \{(a, b) : a, b \in$ e $a = b - 1\}$ não é, visto que 3 R 4 e 4 R 5 não implicam 3 R 5.

Uma relação que é reflexiva, simétrica e transitiva é uma ***relação de equivalência***. Por exemplo, "=" é uma relação de equivalência para os números naturais, mas "<" não é. Se R é uma relação de equivalência para um conjunto A, então, para $a \in A$, a ***classe de equivalência*** de a é o conjunto $[a] = \{b \in A : a\ R\ b\}$, isto é, o conjunto de todos os elementos equivalentes a a. Por exemplo, se definimos $R = \{(a, b) : a, b \in$ e $a + b$ é um número par$\}$, então R é uma relação de equivalência, visto que $a + a$ é par (reflexiva), $a + b$ é par implica $b + a$ é par (simétrica) e $a + b$ é par e $b + c$ é par implicam $a + c$ é par (transitiva). A classe de equivalência de 4 é $[4] = \{0, 2, 4, 6, \ldots\}$ e a classe de equivalência de 3 é $[3] = \{1, 3, 5, 7, \ldots\}$. Apresentamos a seguir um teorema básico de classes de equivalências.

Teorema B.1 (Uma relação de equivalência é o mesmo que uma partição)

As classes de equivalência de qualquer relação de equivalência R para um conjunto A formam uma partição de A, e qualquer partição de A determina uma relação de equivalência para A para a qual os conjuntos na partição são as classes de equivalência.

Prova Para a primeira parte da prova, devemos mostrar que as classes de equivalência de R são conjuntos não vazios, disjuntos aos pares, cuja união é A. Como R é reflexiva, $a \in [a]$, e portanto as classes de equivalência são não vazias;

além disso, visto que todo elemento $a \in A$ pertence à classe de equivalência $[a]$, a união das classes de equivalência é A. Resta mostrar que as classes de equivalência são conjuntos disjuntos aos pares, isto é, se duas classes de equivalência $[a]$ e $[b]$ têm um elemento c em comum, então elas são de fato o mesmo conjunto. Suponha que $a\ R\ c$ e $b\ R\ c$. Por simetria, $c\ R\ b$, e por transitividade, $a\ R\ b$. Assim, para qualquer elemento arbitrário $x \in [a]$, temos $x\ R\ a$ e, por transitividade, $x\ R\ b$ e, assim, $[a] \subseteq [b]$. De modo semelhante, $[b] \subseteq [a]$ e, assim, $[a] = [b]$.

Para a segunda parte da prova, seja $A = \{A_i\}$ uma partição de A e defina $R = \{(a, b)$: existe i tal que $a \in A_i$ e $b \in A_i\}$. Afirmamos que R é uma relação de equivalência em A. A refletividade vale, visto que $a \in A_i$ implica $a\ R\ a$. A simetria vale porque, se $a\ R\ b$, então a e b estão no mesmo conjunto A_i e, por consequência, $b\ R\ a$. Se $a\ R\ b$ e $b\ R\ c$, então os três elementos estão no mesmo conjunto A_i e, assim, $a\ R\ c$ e a transitividade vale. Para verificar que os conjuntos na partição são as classes de equivalência de R, observe que, se $a \in A_i$, então $x \in [a]$ implica $x \in A_i$ e $x \in A_i$ implica $x \in [a]$.

Uma relação binária R para um conjunto A é ***antissimétrica*** se

$a\ R\ b$ e $b\ R\ a$ implicam $a = b$.

Por exemplo, a relação "$\leq$" para os números naturais é antissimétrica, visto que $a \leq b$ e $b \leq a$ implicam $a = b$. Uma relação que é reflexiva, antissimétrica e transitiva é uma ***ordem parcial***, e denominamos um conjunto no qual uma ordem parcial é definida ***conjunto parcialmente ordenado***. Por exemplo, a relação "é um descendente de" é uma ordem parcial no conjunto de todas as pessoas (se considerarmos os indivíduos como sendo seus próprios descendentes).

Em um conjunto parcialmente ordenado A, pode ser que não haja nenhum elemento "máximo" a tal que $b\ R\ a$ para todo $b \in A$. Em vez disso, o conjunto pode conter vários elementos ***maximais*** a tais que, para nenhum $b \in A$, onde $b \neq a$, ocorre que $a\ R\ b$. Por exemplo, uma coleção de caixas de tamanhos diferentes pode conter várias caixas maximais que não cabem dentro de qualquer outra caixa e, apesar disso, não ter nenhuma caixa "máxima" única dentro da qual caberá qualquer outra caixa.[3]

Uma relação R em um conjunto A é uma ***relação total se para todo*** $a, b \in A$, temos a $R\ b$ ou $b\ R\ a$ (ou ambas), isto é, se cada formação de pares de elementos de A está relacionada por R. Uma ordem parcial que é também uma relação total é uma **ordem total** ou **ordem linear**. Por exemplo, a relação "$\leq$" é uma ordem total para os números naturais, mas a relação "é um descendente de" não é uma ordem total para o conjunto de todas as pessoas, visto que existem grupos de indivíduos nos quais nenhum indivíduo descende de outro. Uma relação total que é transitiva, mas não necessariamente antissimétrica, é uma ***pré-ordem*** **total**.

Exercícios

B.2-1 Prove que a relação de subconjunto "$\subseteq$" em todos os subconjuntos de é uma ordem parcial mas não uma ordem total.

B.2-2 Mostre que, para qualquer inteiro positivo n, a relação "equivalente módulo n" é uma relação de equivalência para os inteiros. (Dizemos que a b (mod n) se existe um inteiro q tal que $a - b = qn$.) Em que classes de equivalência essa relação particiona os inteiros?

B.2-3 Dê exemplos de relações que sejam

a. reflexivas e simétricas, mas não transitivas,

b. reflexivas e transitivas, mas não simétricas,

c. simétricas e transitivas, mas não reflexivas.

B.2-4 Seja S um conjunto finito e seja R uma relação de equivalência para $S \times S$. Mostre que, se uma adição R é antissimétrica, então as classes de equivalência de S com relação a R são unitárias.

B.2-5 O professor Narciso afirma que, se uma relação R é simétrica e transitiva, então ela também é reflexiva. Ele oferece a seguinte prova. Por simetria, $a\ R\ b$ implica $b\ R\ a$. Portanto, a transitividade implica $a\ R\ a$. O professor está certo?

B.3 Funções

Dados dois conjuntos A e B, uma ***função*** f é uma relação binária entre A e B tal que, para todo $a \in A$, existe exatamente um $b \in B$ tal que $(a, b) \in f$. O conjunto A é denominado ***domínio*** de f e o conjunto B é denominado ***contradomínio*** de f. Às vezes, escrevemos $f : A \rightarrow B$; e, se $(a, b) \in f$, escrevemos $b = f(a)$, visto que b é determinado unicamente pela escolha de a.

Intuitivamente, a função f atribui um elemento de B a cada elemento de A. A nenhum elemento de A são atribuídos dois elementos diferentes de B, mas o mesmo elemento de B pode ser atribuído a dois elementos diferentes de A. Por exemplo, a relação binária

$$f = \{(a, b) : a, b \in \mathbb{N} \text{ e } b = a \bmod 2\}$$

é uma função $f :\ \ \rightarrow \{0, 1\}$ visto que, para cada número natural a, existe exatamente um valor b em $\{0, 1\}$ tal que $b = a \bmod 2$. Para esse exemplo, $0 = f(0)$, $1 = f(1)$, $0 = f(2)$ etc. Em contraste, a relação binária

$$g = \{(a, b) : a, b \in \mathbb{N} \text{ e } a + b \text{ é par}\}$$

não é uma função, visto que $(1, 3)$ e $(1, 5)$ estão em g e, assim, para a opção $a = 1$, não existe exatamente um b tal que $(a, b) \in g$.

Dada uma função $f : A \rightarrow B$, se $b = f(a)$, dizemos que a é o ***argumento*** de f e que b é o ***valor*** de f em a. Podemos definir uma função declarando seu valor para cada elemento de seu domínio. Por exemplo, poderíamos definir $f(n) = 2n$ para $n \in\ $, o que significa $f = \{(n, 2n) : n \in\ \}$. Duas funções f e g são ***iguais*** se elas têm o mesmo domínio e contradomínio e se, para todo a no domínio, $f(a) = g(a)$.

Uma ***sequência finita*** de comprimento n é uma função f cujo domínio é o conjunto de n inteiros $\{0, 1, \ldots, n - 1\}$. Muitas vezes, denotamos uma sequência finita por uma lista de seus valores: $\langle f(0), f(1), \ldots, f(n - 1)\rangle$. Uma ***sequência infinita*** é uma função cujo domínio é o conjunto dos números naturais. Por exemplo, a sequência de Fibonacci definida pela recorrência (3.22), é a sequência infinita $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\rangle$.

Quando o domínio de uma função f é um produto cartesiano, frequentemente omitimos os parênteses extras que envolvem o argumento de f. Por exemplo, se tivéssemos uma função $f : A_1 \times A_2 \times \ldots \times A_n \rightarrow B$, escreveríamos $b = f(a_1, a_2, \ldots, a_n)$ em vez de $b = f((a_1, a_2, \ldots, a_n))$. Também denominamos cada a_i, um ***argumento*** para a função f, embora tecnicamente o (único) argumento para f seja a n-tupla $(a_1, a_2, \ldots, a_n)$.

Se $f : A \rightarrow B$ é uma função e $b = f(a)$, às vezes, dizemos que b é a ***imagem*** de a sob f. A imagem de um conjunto $A' \subseteq A$ sob f é definida por

$$f(A') = \{b \in B : b = f(a) \text{ para algum } a \in A'\} .$$

A ***imagem*** de f é a imagem do seu domínio, isto é, $f(A)$. Por exemplo, a imagem da função $f :\ \ \rightarrow\ \ $ definida por $f(n) = 2n$ é $f() = \{m : m = 2n$ para algum $n \in\ \}$; em outras palavras, o conjunto de inteiros pares não negativos.

Uma função é uma ***sobrejeção*** se sua imagem é seu contradomínio. Por exemplo, a função $f(n) = n/2$ é uma função sobrejetora de para , visto que todo elemento em aparece como o valor de f para algum argumento. Ao contrário, a função $f(n) = 2n$ não é uma função sobrejetora de para porque nenhum argumento de f pode produzir 3 como valor.

Todavia, a função $f(n) = 2n$ é uma função sobrejetora dos números naturais para os números pares. Uma sobrejeção $f : A \to B$ às vezes é descrita como mapeando A ***sobre*** B. Quando dizemos que f é *sobre*, queremos dizer que ela é sobrejetora.

Uma função $f : A \to B$ é uma ***injeção*** se argumentos distintos de f produzem valores distintos, isto é, se $a \neq a'$ implica $f(a) \neq f(a')$. Por exemplo, a função $f(n) = 2n$ é um função injetora de para , visto que cada número par b é a imagem sob f de, no máximo, um elemento do domínio, isto é, $b/2$. A função $f(n) = n/2$ não é injetora, visto que o valor 1 é produzido por dois argumentos: 2 e 3. Às vezes, uma injeção é denominada função ***um para um***.

Uma função $f : A \to B$ é um ***bijeção*** se é injetora e sobrejetora. Por exemplo, a função $f(n) = (-1)_n\ n/2$ é uma bijeção de para

$$
\begin{aligned}
0 &\to 0, \\
1 &\to -1, \\
2 &\to 1, \\
3 &\to -2, \\
4 &\to 2, \\
&\vdots
\end{aligned}
$$

A função é injetora, já que nenhum elemento de é a imagem de mais do que um elemento de . Ela é sobrejetora, visto que todo elemento de aparece como imagem de algum elemento de . Por consequência, a função é bijetora. Às vezes, uma bijeção é denominada ***correspondência de um para um*** porque forma pares com elementos do domínio e do contradomínio. Uma bijeção de um conjunto A para ele mesmo, às vezes, é denominada ***permutação***.

Quando uma função f é bijetora, definimos sua ***inversa*** f^{-1} como

$$f^{-1}(b) = a \text{ se e somente se } f(a) = b\ .$$

Por exemplo, a inversa da função $f(n) = (-1)^n\ n/2$ é

$$f^{-1} = \begin{cases} 2m & \text{se } m \geq 0, \\ -2m-1 & \text{se } m < 0. \end{cases}$$

Exercícios

B.3-1 Sejam A e B conjuntos finitos e seja $f : A \to B$ uma função. Mostre que

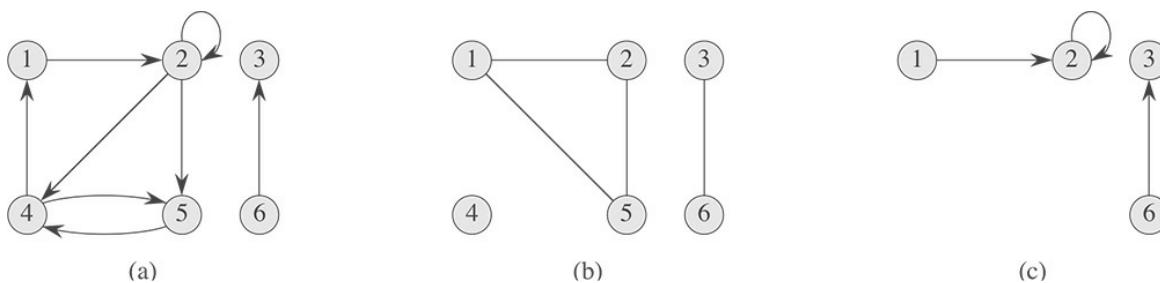
a. se f é injetora, então $|A| \leq |B|$;

b. se f é sobrejetora, então $|A| \geq |B|$.

B.3-2 A função $f(x) = x + 1$ é bijetora quando o domínio e o contradomínio são ? Ela é bijetora quando o domínio e o contradomínio são ?

B.3-3 Dê uma definição natural para a inversa de uma relação binária tal que, se uma relação é de fato uma função bijetora, sua inversa relacional é sua inversa funcional.

B.3-4 ★

Dê uma bijeção de para × .

B.4 GRAFOS

Esta seção apresenta dois tipos de grafos: dirigido e não dirigido. Certas definições encontradas na literatura são diferentes das dadas aqui mas, na maioria das vezes, as diferenças são insignificantes. A Seção 22.1 mostra como representar grafos na memória do computador.

Um ***grafo dirigido*** G é um par (V, E), onde V é um conjunto finito e E é uma relação binária em V. O conjunto V é denominado ***conjunto* de *vértices*** de G, e seus elementos são denominados ***vértices***. O conjunto E é denominado ***conjunto de arestas*** de G, e seus elementos são denominados ***arestas***. A Figura B.2(a) é uma representação pictórica de um grafo dirigido para o conjunto de vértices $\{1, 2, 3, 4, 5, 6\}$. Os vértices são representados por círculos na figura e as arestas são representadas por setas. Observe que são possíveis ***laços*** — arestas de um vértice para ele próprio.

Em um ***grafo não dirigido*** $G = (V, E)$, o conjunto de arestas E consiste em pares de vértices *não ordenados*, em vez de pares ordenados. Isto é, uma aresta é um conjunto $\{u, v\}$, onde $u, v \in V$ e $u \neq v$. Por convenção, usamos a notação (u, v) para uma aresta, em vez da notação de conjuntos $\{u, v\}$, e (u, v) e (v, u) são consideradas a mesma aresta. Em um grafo não dirigido, laços são proibidos e, portanto, toda aresta consiste em dois vértices distintos. A Figura B.2(b) é uma representação pictórica de um grafo não dirigido para o conjunto de vértices $\{1, 2, 3, 4, 5, 6\}$.

Muitas definições para grafos dirigidos e não dirigidos são idênticas, embora certos termos tenham significados ligeiramente diferentes nos dois contextos. Se (u, v) é uma aresta em um grafo dirigido $G = (V, E)$, dizemos que (u, v) é ***incidente do*** vértice u ou ***sai do*** vértice u e é ***incidente no*** vértice v ou ***entra no*** vértice v. Por exemplo, as arestas que saem do vértice 2 na Figura B.2(a) são (2, 2), (2, 4) e (2, 5). As arestas que entram no vértice 2 são (1, 2) e (2, 2). Se (u, v) é uma aresta em um grafo não dirigido $G = (V, E)$, dizemos que (u, v) é ***incidente nos*** vértices u e v. Na Figura B.2(b), as arestas incidentes no vértice 2 são (1, 2) e (2, 5).

Figura B.2 Grafos dirigidos e não dirigidos. **(a)** Um grafo dirigido $G = (V, E)$, onde $V = \{1, 2, 3, 4, 5, 6\}$ e $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. A aresta (2, 2) é um laço. **(b)** Um grafo não dirigido $G = (V, E)$, onde $V = \{1, 2, 3, 4, 5, 6\}$ e $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. O vértice 4 é isolado. **(c)** O subgrafo do grafo da parte (a) induzido pelo conjunto de vértices $\{1, 2, 3, 6\}$.

Se (u, v) é uma aresta em um grafo $G = (V, E)$, dizemos que o vértice v é ***adjacente*** ao vértice u. Quando o grafo é não dirigido, a relação de adjacência é simétrica. Quando o grafo é dirigido, a relação de adjacência não é necessariamente simétrica. Se v é adjacente a u em um grafo dirigido, às vezes, escrevemos $u \to v$. Nas partes (a) e (b) da Figura B.2, o vértice 2 é adjacente ao vértice 1, visto que a aresta (1, 2) pertence a ambos os grafos. O vértice 1 *não* é adjacente ao vértice 2 na Figura B.2(a), visto que a aresta (2, 1) não pertence ao grafo.

O ***grau*** de um vértice em um grafo não dirigido é o número de arestas que nele incidem. Por exemplo, o vértice 2 na Figura B.2(b) tem grau 2. Um vértice cujo grau é 0, como o vértice 4 na Figura B.2(b), é ***isolado***. Em um grafo dirigido, o ***grau de saída*** de um vértice é o número de arestas que saem dele, e o ***grau de entrada*** de um vértice é o número de arestas que entram nele. O ***grau*** de um vértice em um grafo dirigido é seu grau de entrada, mais seu grau de saída. O vértice 2 na Figura B.2(a) tem grau de entrada 2, grau de saída 3 e grau 5.

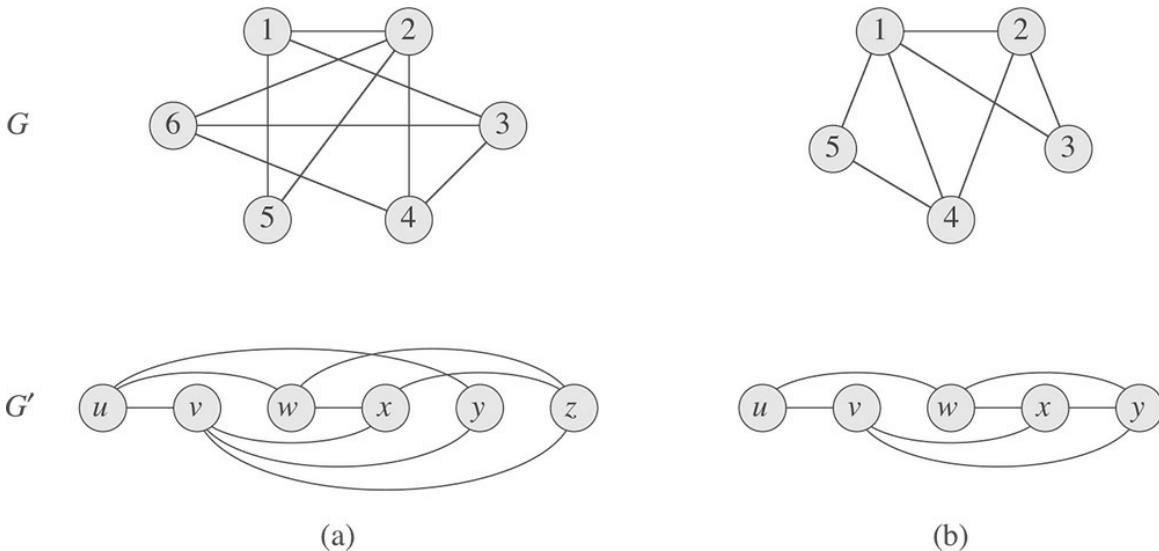
Um ***caminho*** de ***comprimento*** k de um vértice u a um vértice u' em um grafo $G = (V, E)$ é uma sequência $\langle v_0, v_1, v_2, .., v_{kn}\rangle$ de vértices tais que $u = v_0$, $u' = v_k$ e $(v_i{-}1, v_i)$ para $i = 1, 2, \ldots, k$. O comprimento do caminho é o número de arestas no caminho. O caminho ***contém*** os vértices $v_0, v_1, \ldots v_k$ e as arestas $(v_0, v_1), (v_1, v_2), \ldots, (v_k{-}1, v_k)$. (Sempre existe um caminho de comprimento 0 de u até u.) Se existe um caminho p de u até u', dizemos que u' é ***acessível*** a partir de u via p, o que, às vezes, escrevemos como u pu', se G é dirigido. Um caminho é ***simples***[4] se todos os vértices no caminho são distintos. Na Figura B.2(a), o caminho $\langle 1, 2, 5, 4\rangle$ é um caminho simples de comprimento 3. O caminho $\langle 2, 5, 4, 5\rangle$ não é simples.

Um ***subcaminho*** do caminho $p = \langle v_0, v_1, .., v_k\rangle$ é uma subsequência contígua de seus vértices. Isto é, para qualquer $0 \le i \le j \le k$, a subsequência de vértices $\langle v_i, v_i + 1, .., v_j\rangle$ é um subcaminho de p.

Em um grafo dirigido, um caminho $\langle v_0, v_1, .., v_k\rangle$ forma um ***ciclo*** se $v_0 = v_k$ e o caminho contém no mínimo uma aresta. O ciclo é ***simples*** se, além disso, $v_1, v_2, \ldots, v_k$ são distintos. Um laço é um ciclo de comprimento 1. Dois caminhos $\langle v_0, v_1, v_2, .., v_k\text{-}1, v_0\rangle$ e $\langle v_0', v_1', v_2', .., v_k'\text{-}1, v_0'\rangle$ formam o mesmo ciclo se existe um inteiro j tal que $v'^i = v_{(i+j) \bmod k}$ para $i = 0, 1, \ldots, k - 1$. Na Figura B.2(a), o caminho $\langle 1, 2, 4, 1\rangle$ forma o mesmo ciclo que os caminhos $\langle 2, 4, 1, 2\rangle$ e $\langle 4, 1, 2, 4\rangle$. Esse ciclo é simples, mas o ciclo $\langle 1, 2, 4, 5, 4, 1\rangle$ não é. O ciclo $\langle 2, 2\rangle$ formado pela aresta (2, 2) é um laço. Um grafo dirigido sem nenhum laço é ***simples***. Em um grafo não dirigido, um caminho $\langle v_0, v_1, .., v_k\rangle$ forma um ***ciclo*** se $v_0 = v_k$ e todas suas arestas são distintas; o ciclo é simples se $v_1, v_2, \ldots, v_k$ são distintos. Por exemplo, na Figura B.2(b), o caminho $\langle 1, 2, 5, 1\rangle$ é um ciclo. Um grafo que não tenha nenhum ciclo simples é ***acíclico.***

Um grafo não dirigido é ***conexo*** se todo vértice pode ser alcançado de todos os outros vértices. As ***componentes conexas*** de um grafo não dirigido são as classes de equivalência de vértices sob a relação "pode ser alcançado de". O grafo da Figura B.2(b) tem três componentes conexas:

{1, 2, 5}, {3, 6} e {4}. Todo vértice em {1, 2, 5} pode ser alcançado de cada um dos outros vértices em {1, 2, 5}. Um grafo não dirigido é conexo se tem exatamente uma componente conexa. As arestas de uma componente conexa são as que incidem somente nos vértices da componente; em outras palavras, a aresta (u, v) é uma aresta de uma componente conexa se e somente se u e v são vértices da componente.

Um grafo dirigido é ***fortemente conexo*** se cada vértice pode ser alcançado de qualquer outro vértice. As ***componentes fortemente conexas*** de um grafo dirigido são as classes de equivalência de vértices sob a relação "são mutuamente acessíveis". Um grafo dirigido é fortemente conexo se tem somente uma componente fortemente conexa. O grafo da Figura B.2(a) tem três componentes fortemente conexas: {1, 2, 4, 5}, {3} e {6}. Todos os pares de vértices em {1, 2, 4, 5} são mutuamente acessíveis. Os vértices {3, 6} não formam uma componente fortemente conexa, visto que o vértice 6 não pode ser alcançado do vértice 3.

Dois grafos $G = (V, E)$ e $G' = (V', E')$ são ***isomorfos*** se existe uma bijeção $f : V \to V'$ tal que $(u, v) \in E$ se e somente se $(f(u), f(v)) \in E'$. Em outras palavras, podemos renomear os vértices de G como vértices de G', mantendo as arestas correspondentes em G e G'. A Figura B.3(a) mostra um par de grafos isomorfos G e G' com conjuntos de vértices respectivos $V = \{1, 2, 3, 4, 5, 6\}$ e $V' = \{u, v, w, x, y, z\}$. O mapeamento de V para V' dado por $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$ dá a função bijetora requerida. Os grafos na Figura B.3(b) não são isomorfos. Embora ambos tenham cinco vértices e sete arestas, o grafo na parte superior da figura tem um vértice de grau 4, e o grafo na parte inferior, não.

Dizemos que um grafo $G' = (V', E')$ é um ***subgrafo*** de $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$. Dado um conjunto $V' \subseteq V$, o subgrafo de G ***induzido*** por V' é o grafo $G' = (V', E')$, onde

$$E' = \{(u, v) \in \mathrm{E} : u, v \in V'\}.$$

O subgrafo induzido pelo conjunto de vértices {1, 2, 3, 6} na Figura B.2(a) aparece na Figura B.2(c) e tem o conjunto de arestas {(1, 2), (2, 2), (6, 3)}.

Dado um grafo não dirigido $G = (V, E)$, a ***versão dirigida*** de G é o grafo dirigido $G' = (V, E')$, onde $(u, v) \in E'$ se e somente se $(u, v) \in E$. Isto é, substituímos cada aresta não dirigida (u, v) em G pelas duas arestas dirigidas (u, v) e (v, u) na versão dirigida. Dado um grafo dirigido $G = (V, E)$, a ***versão não dirigida*** de G é o grafo não dirigido $G' = (V, E')$, onde $(u, v) \in E'$ se e somente se $u \neq v$ e E contem pelo menos uma das arestas (u, v) e (v, u). Isto é, a versão não dirigida contém as arestas de G "com suas direções eliminadas" e com laços também eliminados. (Como (u, v) e (v, u) são a mesma aresta em um grafo não dirigido, a versão não dirigida de um grafo dirigido a contém somente uma vez, mesmo que o grafo dirigido contenha as arestas (u, v) e (v, u).) Em um grafo dirigido $G = (V, E)$, um ***vizinho*** de um vértice u é qualquer vértice adjacente a u na versão não dirigida de G. Isto é, v é um vizinho de u se $u \neq v$ e, ou $(u, v) \in E$ ou $(v, u) \in E$. Em um grafo não dirigido, u e v são vizinhos se são adjacentes.

Figura B.3 (a) Um par de grafos isomorfos. Os vértices do grafo na parte superior da figura são mapeados para os vértices do grafo na parte inferior da figura por $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$. **(b)** Dois grafos que não são isomorfos, visto que o grafo na parte superior da figura tem um vértice de grau 4 e o grafo na parte inferior da figura não tem.

Vários tipos de grafos têm nomes especiais. Um ***grafo completo*** é um grafo não dirigido no qual todo par de vértices é adjacente. Um ***grafo bipartido*** é um grafo não dirigido $G = (V, E)$ no qual V pode ser particionado em dois conjuntos V_1 e V_2 tais que $(u, v) \in E$ implica ou $u \in V_1$ e $v \in V_2$ ou $u \in V_2$ e $v \in V_1$. Isto é, todas as arestas ficam entre os dois conjuntos V_1 e V_2. Um grafo acíclico não dirigido é uma ***floresta***, e um grafo conexo acíclico não dirigido é uma ***árvore*** (***livre***) (veja a Seção B.5). Muitas vezes, tomamos as primeiras letras de "grafo acíclico dirigido" e denominamos tal grafo ***gad***.

Há duas variantes de grafos que você poderá encontrar ocasionalmente. Um ***multigrafo*** é semelhante a um grafo não dirigido, mas pode ter várias arestas entre os mesmos vértices e laços. Um ***hipergrafo*** é semelhante a um grafo não dirigido, mas cada ***hiperaresta***, em vez de conectar dois vértices, conecta um subconjunto arbitrário de vértices. Muitos algoritmos escritos para grafos dirigidos e não dirigidos comuns podem ser adaptados para funcionar nessas estruturas semelhantes a grafos.

A ***contração*** de um grafo não dirigido $G = (V, E)$ por uma aresta $e = (u, v)$ é um grafo $G' = (V', E')$, onde $V' = V - \{u, v\} \cup \{x\}$ e x é um novo vértice. O conjunto de arestas E' é formado por E, sem a aresta (u, v) e, para cada vértice w adjacente a u ou v, é eliminada a aresta (u, w) ou a aresta (v, w), dependendo de qual está em E, e em seu lugar é adicionado a nova aresta (x, w). Na prática, u e n são "contraídos"a um vértice só.

Exercícios

B.4-1 Os participantes de uma festa de professores de uma faculdade cumprimentam-se com um aperto de mão e cada professor memoriza quantas pessoas cumprimentou. No final da festa, o chefe do departamento soma o número de vezes que cada professor cumprimentou os outros. Mostre que o resultado é par, provando o ***lema do cumprimento***: se $G = (V, E)$ é um grafo não dirigido, então

$$\sum_{v \in V} \text{grau}(v) = 2\,|E|\,.$$

B.4-2 Mostre que, se um grafo dirigido ou não dirigido contém um caminho entre dois vértices u e v, contém um caminho simples entre u e v. Mostre que, se um grafo dirigido contém um ciclo, então contém um ciclo simples.

B.4-3 Mostre que qualquer grafo conexo não dirigido $G = (V, E)$ satisfaz $|E| \geq |V| - 1$.

B.4-4 Verifique que em um grafo não dirigido, a relação "pode ser alcançado de" é uma relação de equivalência para os vértices do grafo. Qual das três propriedades de uma relação de equivalência é válida em geral para a relação "pode ser alcançado de" para os vértices de um grafo dirigido?

B.4-5 Qual é a versão não dirigida do grafo dirigido na Figura B.2(a)? Qual é a versão dirigida do grafo não dirigido na Figura B.2(b)?

B.4-6 ★

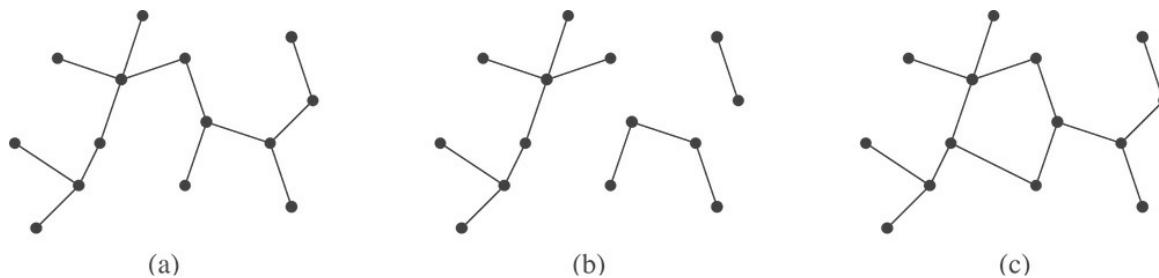
Mostre que podemos representar um hipergrafo por um grafo bipartido sempre que a incidência no hipergrafo corresponda à adjacência no grafo bipartido. (*Sugestão*: Faça com que um conjunto de vértices no grafo bipartido corresponda a vértices do hipergrafo e com que o outro conjunto de vértices do grafo bipartido corresponda a hiperarestas.)

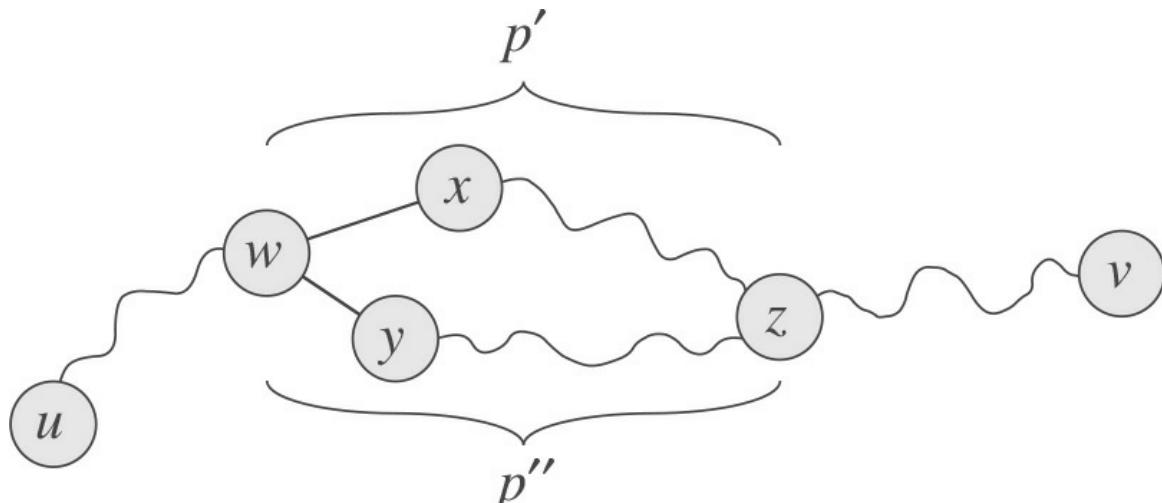
B.5 ÁRVORES

Do mesmo modo que para os grafos, há muitas noções de árvores relacionadas, embora ligeiramente diferentes. Esta seção apresenta definições e propriedades matemáticas de vários tipos de árvores. As Seções 10.4 e 22.1 descrevem como representamos árvores na memória de computadores.

B.5.1 ÁRVORES LIVRES

Como definimos na Seção B.4, uma ***árvore livre*** é um grafo acíclico conexo não dirigido. Muitas vezes, omitimos o adjetivo "livre" quando dizemos que um grafo é uma árvore. Se um grafo não dirigido é acíclico mas possivelmente desconexo, é uma ***floresta***. Muitos algoritmos que funcionam para árvores também funcionam para florestas. A Figura B.4(a) mostra uma árvore livre e a Figura B.4(b) mostra uma floresta. A floresta na Figura B.4(b) não é uma árvore porque não é conexa. O grafo na Figura B.4(c) não é nem uma árvore nem uma floresta porque contém um ciclo.

O teorema a seguir abrange muitos fatos importantes sobre árvores livres.

Teorema B.2 (*Propriedades de árvores livres*)

Seja $G = (V, E)$ um grafo não dirigido. As afirmativas a seguir são equivalentes.

1. G é uma árvore livre.
2. Quaisquer dois vértices em G estão ligados por um caminho simples único.
3. G é conexo, mas, se qualquer aresta for eliminada de E, o grafo resultante é desconexo.
4. G é conexo e $|E| = |V| - 1$.
5. G é acíclico e $|E| = |V| - 1$.
6. G é acíclico, mas, se qualquer aresta for adicionada a E, o grafo resultante contém um ciclo.

Prova (1) ⇒ (2): Dado que uma árvore é conexa, quaisquer dois vértices em G estão ligados por no mínimo um caminho simples. Suponha, por contradição, que os vértices u e v estão ligados por dois caminhos simples distintos p_1 e p_2, como mostra a Figura B.5. Seja w o vértice no qual os caminhos divergem pela primeira vez; isto é, w é o primeiro vértice em p_1 e também em p_2 cujo sucessor em p_1 é x e cujo sucessor em p_2 é y, onde $x \neq y$. Seja z o primeiro vértice no qual os caminhos reconvergem; isto é, z é o primeiro vértice que vem depois de w em p_1 que também está em p_2. Seja p' o subcaminho de p_1 que parte de w e passa por x e chega até z, e seja p'' o subcaminho de p_2 que parte de w e passa por y e chega até z. Os caminhos p' e p'' não compartilham nenhum vértice, exceto suas extremidades. Assim, o caminho obtido pela concatenação de p' com o inverso de p'' é um ciclo, o que contradiz nossa hipótese de que G é uma árvore. Assim, se G é uma árvore, pode haver, no máximo, um caminho simples entre dois vértices.

(2)⇒ (3): Se quaisquer dois vértices em G estão ligados por um caminho simples único, então G é conexo. Seja (u, v) qualquer aresta em E. Essa aresta é um caminho de u a v e, portanto, deve ser o caminho único de u até v. Se eliminarmos (u, v) de G, não há nenhum caminho de u a v e, consequentemente, sua eliminação torna G desconexo.

Figura B.4 (a) Uma árvore livre. **(b)** Uma floresta. **(c)** Um grafo que contém um ciclo e que, portanto, não é nem uma árvore nem uma floresta.

Figura B.5 Uma etapa na prova do Teorema B.2: se (1) G é uma árvore livre, então (2) quaisquer dois vértices em G estão ligados por um caminho simples único. Considere, por contradição, que os vértices u e v são ligados por dois caminhos simples distintos p_1 e p_2. Esses

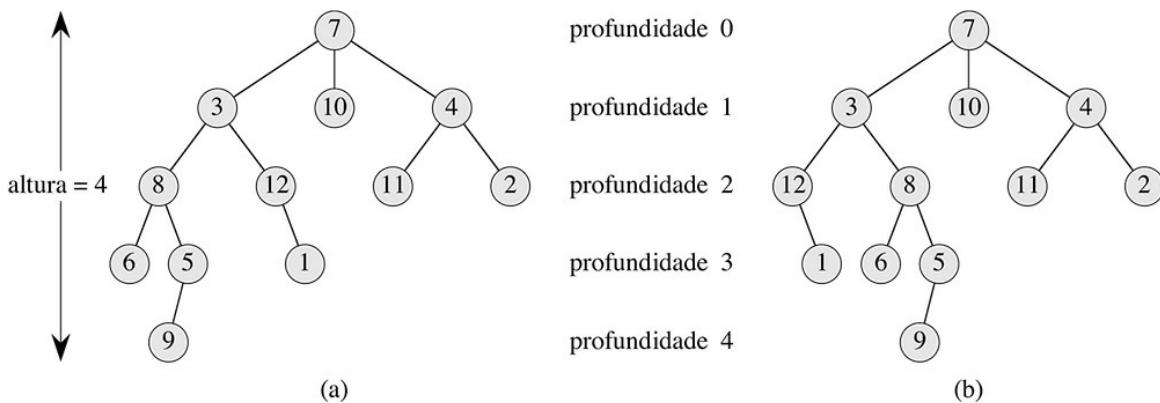
caminhos divergem primeiro no vértice w e depois reconvergem primeiro no vértice z. O caminho p' concatenado com o inverso do caminho p'' forma um ciclo, o que produz a contradição.

(3)⇒ (4): Por hipótese, o grafo G é conexo e, pelo Exercício B.4-3, temos $|E| \geq |V| - 1$. Provaremos $|E| \leq |V| - 1$ por indução. Um grafo conexo com $n = 1$ ou $n = 2$ vértices tem $n - 1$ arestas. Suponha que G tenha $n \leq 3$ vértices e que todos os grafos que satisfazem (3) com menos de n vértices também satisfazem $|E| \leq |V| - 1$. Eliminar uma aresta arbitrária de G separa o grafo em $k \geq 2$ componentes conexas (na realidade, $k = 2$). Cada componente satisfaz (3) ou, do contrário, G não satisfaria (3). Se virmos cada componente conexa V_i, com conjunto de arestas E_i, como sua própria árvore livre, então, visto que cada componente tem menos do que $|V|$ vértices, pela hipótese de indução temos $|E_i| \leq |V_i| - 1$. Assim, o número de arestas em todas as componentes combinadas é, no máximo, $|V| - k \leq |V| - 2$. Adicionar a aresta eliminada produz $|E| \leq |V| - 1$.

(4) ⇒ (5): Suponha que G seja conexo e que $|E| = |V| - 1$. Devemos mostrar que G é acíclico. Suponha que G tenha um ciclo que contém k vértices $v_1, v_2, \ldots, v_k$ e, sem perda da generalidade, suponha que esse ciclo seja simples. Seja $G_k = (V_k, E_k)$ o subgrafo de G que consiste no ciclo. Observe que $|V_k| = |E_k| = k$. Se $k < |V|$, deve existir um vértice $v_{k+1} \in V - V_k$ que é adjacente a algum vértice $v_i \in V_k$, visto que G é conexo. Defina $G_{k+1} = (V_{k+1}, E_{k+1})$ como o subgrafo de G com $V_{k+1} = V_k \cup \{v_{k+1}\}$ e $E_{k+1} = E_k \cup \{v_i, v_{k+1}\}$. Observe que $|V_{k+1}| = |E_{k+1}| = k + 1$. Se $k + 1 < |V|$, podemos continuar, definindo G_{k+2} da mesma maneira, e assim por diante, até obtermos $G_n = (V_n, E_n)$, onde $n = |V|$, $V_n = V$ e $|E_n| = |V_n| = |V|$. Como G_n é um subgrafo de G, temos $E_n \subseteq E$ e, portanto, $|E| \leq |V|$, o que contradiz a hipótese de que $|E| = |V| - 1$. Assim, G é acíclico.

(5)⇒ (6): Suponha que G seja acíclico e que $|E| = |V| - 1$. Seja k o número de componentes conexas de G. Cada componente conexa é uma árvore livre por definição e, visto que (1) implica (5), a soma de todas as arestas em todas as componentes conexas de G é $|V| - k$. Consequentemente, devemos ter $k = 1$, e G é de fato uma árvore. Visto que (1) implica (2), quaisquer dois vértices em G estão ligados por um caminho simples único. Portanto, adicionar qualquer aresta a G cria um ciclo.

(6) ⇒ (1): Suponha que G seja acíclico, mas que adicionar qualquer aresta a E cria um ciclo. Devemos mostrar que G é conexo. Sejam u e v vértices arbitrários em G. Se u e v ainda não forem adjacentes, adicionar a aresta (u, v) cria um ciclo no qual todas as arestas com exceção de (u, v) pertencem a G. Assim, o ciclo menos a aresta (u, v) deve conter um caminho de u a v e, visto que u e v foram escolhidos arbitrariamente, G é conexo.

B.5.2 ÁRVORES ENRAIZADAS E ÁRVORES ORDENADAS

Uma ***árvore enraizada*** é uma árvore livre na qual um dos vértices é destacado. Denominamos ***raiz*** da árvore esse vértice distinto. Muitas vezes nos referimos a um vértice de uma árvore enraizada como um ***nó***[5] da árvore. A Figura B.6(a) mostra uma árvore enraizada em um conjunto de 12 nós com raiz 7.

Considere um nó x em uma árvore enraizada T com raiz r. Denominamos qualquer nó y no caminho simples único de r a x por ***ancestral*** de x. Se y é um ancestral de x, então x é um ***descendente*** de y. (Todo nó é ao mesmo tempo um ancestral e um descendente de si mesmo.) Se y é um ancestral de x e $x \neq y$, então y é um ***ancestral próprio*** de x, e x é um ***descendente próprio*** de y. A ***subárvore enraizada em x*** é a árvore induzida por descendentes de x com raiz em x. Por exemplo, a subárvore enraizada no nó 8 na Figura B.6(a) contém os nós 8, 6, 5 e 9.

Se a última aresta no caminho simples da raiz r de uma árvore T a um nó x é (y, x), então y é o ***pai*** de x, e x é um ***filho*** de y. A raiz é o único nó em T que não tem nenhum pai. Se dois nós têm o mesmo pai, eles são ***irmãos***. Um nó sem nenhum filho é uma ***folha*** ou um ***nó externo***. Um nó que não é uma folha é um ***nó interno***.

O número de filhos de um nó x em uma árvore enraizada T é denominado ***grau*** de x.[6] O comprimento do caminho simples da raiz r a um nó x é a ***profundidade*** de x em T. Um ***nível*** de uma árvore consiste em todos os nós que estão na mesma profundidade. A ***altura*** de um nó em uma árvore é o número de arestas no caminho simples descendente mais longo do nó a uma folha, e a altura de uma árvore é a altura de sua raiz. A altura de uma árvore também é igual à maior profundidade de qualquer nó na árvore.

Uma ***árvore ordenada*** é uma árvore enraizada na qual os filhos de cada nó estão ordenados. Isto é, se um nó tem k filhos, então existe um primeiro filho, um segundo filho, … e um k-ésimo filho. As duas árvores na Figura B.6 são diferentes quando consideradas como árvores ordenadas, mas são idênticas quando consideradas apenas como árvores enraizadas.

Figura B.6 Árvores enraizadas e árvores ordenadas. **(a)** Uma árvore enraizada com altura 4. A árvore é desenhada de um modo padrão: a raiz (nó 7) está na parte superior, seus filhos (os nós com profundidade 1) estão abaixo dela, os filhos de seus filhos (nós com profundidade 2) estão abaixo destes, e assim por diante. Se a árvore é ordenada, a ordem relativa da esquerda para a direita dos filhos de um nó é importante; caso contrário, não é importante. **(b)** Outra árvore enraizada. Como uma árvore enraizada, ela é idêntica à árvore em (a), mas, como uma árvore ordenada, ela é diferente, visto que os filhos do nó 3 aparecem em uma ordem diferente.

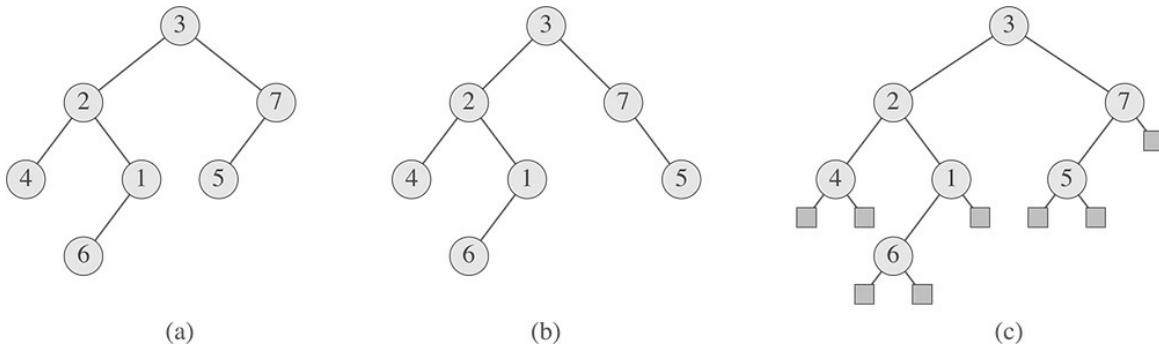
B.5.3 ÁRVORES BINÁRIAS E ÁRVORES POSICIONAIS

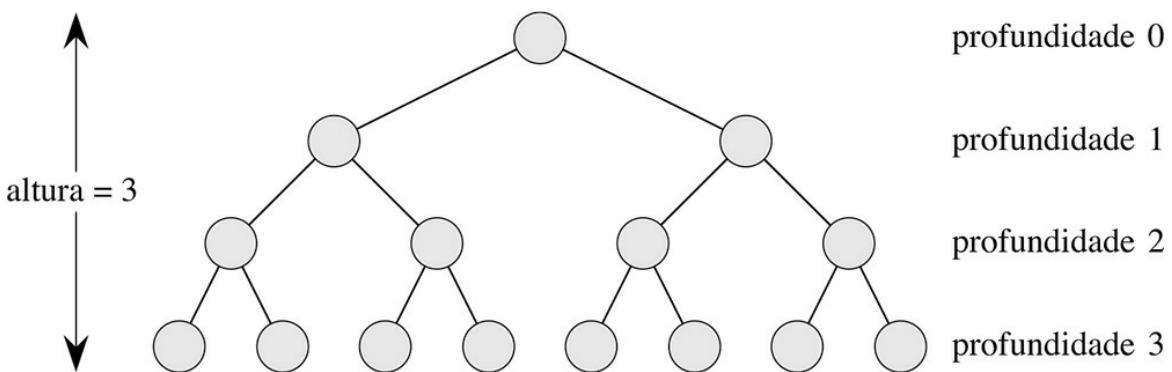
Definimos árvores binárias recursivamente. Uma ***árvore binária*** T é uma estrutura definida para um conjunto finito de nós que

- não contém nenhum nó ou
- é composta por três conjuntos disjuntos de nós: um nó ***raiz***, uma árvore binária denominada sua ***subárvore da esquerda*** e uma árvore binária denominada sua ***subárvore da direita***.

A árvore binária que não contém nenhum nó é denominada ***árvore vazia*** ou ***árvore nula***, às vezes, denotada por NIL. Se a subárvore da esquerda é não vazia, sua raiz é denominada ***filho da esquerda*** da raiz da árvore inteira. Da mesma forma, a raiz de uma subárvore da direita não nula é o ***filho da direita*** da raiz da árvore inteira. Se uma subárvore é a árvore nula NIL, dizemos que o filho está ***ausente*** ou está ***faltando***. A Figura B.7(a) mostra uma árvore binária.

Uma árvore binária não é simplesmente uma árvore ordenada na qual cada nó tem, no máximo, grau 2. Por exemplo, em uma árvore binária, se um nó tem apenas um filho, a posição do filho — seja ele o ***filho da esquerda*** ou o ***filho da direita*** — é importante. Em uma árvore ordenada, não há como distinguir se um filho isolado é um filho da esquerda ou da direita. A Figura B.7(b) mostra uma árvore binária que é diferente da árvore na Figura B.7(a) por causa da posição de um único nó. Contudo, se consideradas como árvores ordenadas, as duas árvores são idênticas.

Podemos representar as informações de posicionamento em uma árvore binária pelos nós internos de uma árvore ordenada, como mostra a Figura B.7(c). A ideia é substituir cada filho que falta na árvore binária por um nó que não tem nenhum filho. Esses nós de folha estão desenhados como quadrados na figura. A árvore resultante é uma ***árvore binária cheia***: cada nó ou é uma folha ou tem grau exatamente 2. Não há nós de grau 1. Consequentemente, a ordem dos filhos de um nó preserva as informações de posição.

Podemos estender as informações de posicionamento que distinguem árvores binárias de árvores ordenadas a árvores com mais de dois filhos por nó. Em uma ***árvore posicional***, os filhos de um nó são identificados por números

inteiros positivos distintos. O i-ésimo filho de um nó é ***ausente*** se nenhum filho é identificado com o inteiro i. Uma árvore ***k-ária*** é uma árvore posicional na qual, para todo nó, todos os filhos com rótulos maiores que k estão faltando. Assim, uma árvore binária é uma árvore k-ária com $k = 2$.

Uma ***árvore k-ária completa*** é uma árvore k-ária na qual todas as folhas têm a mesma profundidade e todos os nós internos têm grau k. A Figura B.8 mostra uma árvore binária completa de altura 3. Quantas folhas tem uma árvore k-ária completa de altura h? A raiz tem k filhos na profundidade 1, cada um deles tem k filhos na profundidade 2, e assim por diante. Então, o número de folhas na profundidade h é k_h. Consequentemente, a altura de uma árvore k-ária completa com n folhas é $\log_k n$. O número de nós internos de uma árvore k-ária completa de altura h é

$$\begin{aligned} 1+k+k^2+\cdots+k^{h-1} &= \sum_{i=0}^{h-1} k^i \\ &= \frac{k^h-1}{k-1} \end{aligned}$$

pela equação (A.5). Assim, uma árvore binária completa tem $2^h - 1$ nós internos.

Figura B.7 Árvores binárias. **(a)** Uma árvore binária desenhada de um modo padrão. O filho da esquerda de um nó é desenhado abaixo e à esquerda do nó. O filho da direita é desenhado abaixo e à direita do nó. **(b)** Uma árvore binária diferente da que está em (a). Em (a), o filho da esquerda do nó 7 é 5, e o filho da direita está ausente. Em **(b)**, o filho da esquerda do nó 7 está ausente e o filho da direita é 5. Como árvores ordenadas, essas árvores são idênticas, mas, como árvores binárias, elas são distintas. **(c)** A árvore binária em (a) representada pelos nós internos de uma árvore binária cheia: uma árvore ordenada na qual cada nó interno tem grau 2. As folhas na árvore são mostradas como quadrados.

Figura B.8 Uma árvore binária completa de altura 3 com oito folhas e sete nós internos.

Exercícios

B.5-1 Desenhe todas as árvores livres compostas pelos três vértices x, y e z. Desenhe todas as árvores enraizadas com nós x, y e z que têm x como raiz. Desenhe todas as árvores ordenadas com nós x, y e z que têm x como raiz. Desenhe todas as árvores binárias com nós x, y e z que têm x como raiz.

B.5-2 Seja $G = (V, E)$ um grafo acíclico dirigido no qual existe um vértice $v_0 \in V$ tal que existe um caminho único de v_0 até todo vértice $v \in V$. Prove que a versão não dirigida de G forma uma árvore.

B.5-3 Mostre por indução que o número de nós de grau 2 em qualquer árvore binária não vazia é uma unidade menor que o número de folhas. Conclua que o número de nós internos em uma árvore binária cheia é uma unidade menor que o número de folhas.

B.5-4 Use indução para mostrar que uma árvore binária não vazia com n nós tem altura de, no mínimo, $\lg n$.

B.5-5 ★

O ***comprimento de caminho interno*** de uma árvore binária cheia é a soma, aplicada a todos os nós internos da árvore, das profundidades de cada nó. De modo semelhante, o ***compri- mento de caminho externo*** é a soma, aplicada a todas as folhas da árvore, das profundidades de cada folha. Considere uma árvore binária cheia com n nós internos, comprimento de caminho interno i e comprimento de caminho externo e. Prove que $e = i + 2n$.

B.5-6 ★

Vamos associar um "peso" $w(x) = 2^{-d}$ a cada folha x de profundidade d em uma árvore binária T e seja L o conjunto de folhas de T. Prove que $\Sigma_{x \in L} w(x) \leq 1$. (Isso é conhecido como ***desigualdade de Kraft***.)

B.5-7 ★

Mostre que, se $L \geq 2$, então toda árvore binária com L folhas contém uma subárvore que tem entre $L/3$ e $2L/3$ folhas, inclusive.

Problemas

B-1 ***Coloração de grafos***

Dado um grafo não dirigido $G = (V, E)$, uma ***k-coloração*** é uma função $c : V \rightarrow \{0, 1, \ldots, k-1\}$ tal que $c(u) \neq c(v)$ para toda aresta $(u, v) \in E$. Em outras palavras, os números $0, 1, \ldots, k-1$ representam as k cores, e vértices adjacentes devem ter cores diferentes.

a. Mostre que qualquer árvore pode ser 2-colorida.

b. Mostre que os itens seguintes são equivalentes:

1. G é bipartido.

2. G pode ser 2-colorido.

3. G não tem nenhum ciclo de comprimento ímpar.

c. Seja d o grau máximo de qualquer vértice em um grafo G. Prove que podemos colorir G com $d + 1$ cores.

d. Mostre que, se G tem $O(|V|)$ arestas, então G pode ser colorido com $O(\sqrt{|V|})$ cores.

B-2 ***Grafos de amigos***

Reescreva cada uma das declarações a seguir como um teorema para grafos não dirigidos e depois prove o teorema. Considere que amizade é simétrica, mas não reflexiva.

a. Qualquer grupo de no mínimo duas pessoas contém no mínimo duas pessoas com o mesmo número de amigos no grupo.

b. Todo grupo de seis pessoas contém no mínimo três amigos mútuos ou no mínimo três estranhos mútuos.

c. Qualquer grupo de pessoas pode ser repartido em dois subgrupos tais que no mínimo metade dos amigos de cada pessoa pertence ao subgrupo do qual essa pessoa *não* é um membro.

d. Se toda pessoa em um grupo é amiga de no mínimo metade das pessoas no grupo, então o grupo pode se sentar em torno de uma mesa de tal modo que toda pessoa fique sentada entre dois amigos.

B-3 ***Bisseção de árvores***

Muitos algoritmos de divisão e conquista aplicáveis a grafos exigem que o grafo seja dividido em dois subgrafos de tamanhos aproximadamente iguais, que são induzidos por uma partição dos vértices. Este problema investiga a bisseção de árvores formadas pela eliminação de um pequeno número de arestas. Exige-se, sempre que dois vértices acabam na mesma subárvore após a eliminação de arestas, que estejam na mesma partição.

a. Mostre que podemos particionar os vértices de qualquer árvore binária de n vértices em dois conjuntos A e B, tais que $|A| \leq 3n/4$ e $|B| \leq 3n/4$, eliminando uma única aresta.

b. Mostre que a constante 3/4 na parte (a) é ótima no pior caso dando um exemplo de árvore binária simples cuja partição de equilíbrio mais uniforme obtida com a eliminação de uma única aresta tenha $|A| = 3n/4$.

c. Mostre que, removendo no máximo $O(\lg n)$ arestas, podemos particionar os vértices de qualquer árvore binária de n vértices em dois conjuntos A e B tais que $|A| = n/2$ e $|B| = n/2$.

NOTAS DO CAPÍTULO

G. Boole foi o pioneiro no desenvolvimento da lógica simbólica e introduziu muitas das notações básicas de conjuntos em um livro publicado em 1854. A moderna teoria dos conjuntos foi criada por G. Cantor durante o período de 1874 a 1895. Cantor focalizou principalmente os conjuntos de cardinalidade infinita. O termo "função" é atribuído a G. W. Leibniz, que o usou para se referir a várias espécies de fórmulas matemáticas. Sua definição limitada foi generalizada várias vezes. A teoria dos grafos teve origem em 1736, quando L. Euler provou que era impossível cruzar cada uma das sete pontes da cidade de Königsberg exatamente uma vez e retornar ao ponto de partida.

O livro de Harary [160] é um compêndio útil de muitas definições e resultados da teoria dos grafos.

---

[1]Uma variação de um conjunto, que pode conter o mesmo objeto mais de uma vez, é denominada ***multiconjunto.***

[2]Alguns autores começam os números naturais com 1 em vez de 0. A tendência moderna parece ser começar com 0.

[3] Para sermos exatos, para que a relação "caber dentro de" seja uma ordem parcial, precisamos adotar que uma caixa cabe dentro dela mesma.

[4]Alguns autores referem-se ao que denominamos caminho como um "passeio" e ao que denominamos caminho simples como "caminho" apenas. Usamos os termos "caminho" e "caminho simples" em todo este livro de modo compatível com suas definições.

[5] O termo “nó” é muito usado na literatura da teoria dos grafos como sinônimo de “vértice”. Reservaremos o termo “nó” para indicar um vértice de uma árvore enraizada.

[6] Observe que o grau de um nó depende de considerarmos T como uma árvore enraizada ou como uma árvore livre. O grau de um vértice em uma árvore livre é, como em qualquer grafo não dirigido, o número de vértices adjacentes. Porém, em uma árvore enraizada, o grau é o número de filhos — o pai de um nó não conta para definir seu grau.

C CONTAGEM E PROBABILIDADE

Este capítulo faz uma revisão da análise combinatória elementar e da teoria da probabilidade. Se o leitor tiver um bom conhecimento nessas áreas, basta ler rapidamente o início do capítulo e se concentrar nas últimas seções. A maior parte dos capítulos deste livro não requer que o leitor conheça probabilidade, mas para alguns capítulos tal conhecimento é essencial.

A Seção C.1 faz uma revisão dos resultados elementares de teoria da contagem, incluindo fórmulas padrões para contagem de permutações e combinações. Os axiomas da probabilidade e os fatos básicos relativos a distribuições de probabilidade são apresentados na Seção C.2. Variáveis aleatórias são introduzidas na Seção C.3, juntamente com as propriedades de esperança e variância. A Seção C.4 examina as distribuições geométricas e binomiais que surgem do estudo de tentativas de Bernoulli. O estudo da distribuição binomial continua na Seção C.5, uma discussão avançada das "caudas" da distribuição.

C.1 CONTAGEM

A teoria da contagem tenta responder à pergunta "quantos(as)?" sem realmente enumerar todas as escolhas. Por exemplo, poderíamos perguntar: "Quantos números diferentes de n bits existem?" ou "Quantas ordenações de n elementos distintos existem?" Nesta seção, faremos uma revisão dos elementos da teoria da contagem. Visto que parte do material pressupõe uma compreensão básica de conjuntos, aconselhamos o leitor a começar pela revisão do material na Seção B.1.

Regras da soma e do produto

Ás vezes, podemos expressar um conjunto de itens que desejamos contar como uma união de conjuntos disjuntos ou como um produto cartesiano de conjuntos.

A ***regra da soma*** afirma que o número de modos de escolher um elemento de um entre dois conjuntos *disjuntos* é a soma das cardinalidades dos conjuntos. Ou seja, se A e B são dois conjuntos finitos sem nenhum membro em comum, então $|A \cup B| = |A| + |B|$, que decorre da equação (B.3). Por exemplo, cada posição na placa de um automóvel é uma letra ou um dígito. Portanto, o número de possibilidades para cada posição é $26 + 10 = 36$, já que existem 26 escolhas se for uma letra e 10 escolhas se for um dígito.

A ***regra do produto*** afirma que o número de modos de escolher um par ordenado é o número de modos de escolher o primeiro elemento vezes o número de modos de escolher o segundo elemento. Isto é, se A e B são dois conjuntos finitos, então $|A \cdot B| = |A| \cdot |B|$, que é simplesmente a equação (B.4). Por exemplo, se uma sorveteria oferece 28 sabores de sorvete e quatro coberturas, o número de *sundaes* possíveis com uma bola de sorvete e uma cobertura é $28 \cdot 4 = 112$.

Cadeias

Uma ***cadeia*** em um conjunto finito S é uma sequência de elementos de S. Por exemplo, há oito cadeias binárias de comprimento 3:

$$000, 001, 010, 011, 100, 101, 110, 111 .$$

Às vezes, denominamos uma cadeia de comprimento k ***k-cadeia***. Uma ***subcadeia*** s' de um cadeia s é uma sequência ordenada de elementos consecutivos de s. Uma ***k-subcadeia*** de uma cadeia é uma subcadeia de comprimento k. Por exemplo, 010 é uma 3-subcadeia de 01101001 (a 3-subcadeia que começa na posição 4), mas 111 não é uma subcadeia de 01101001.

Podemos ver uma k-cadeia em um conjunto S como um elemento do produto cartesiano S_k de tuplas de k elementos; assim, existem $|S|^k$ cadeias de comprimento k. Por exemplo, o número de cadeias binárias de k elementos é 2^k. Intuitivamente, para construir uma cadeia de k elementos em um conjunto de n elementos, temos n modos de escolher o primeiro elemento; para cada uma dessas opções, temos n modos de escolher o segundo elemento, e assim por diante k vezes. Essa construção conduz ao produto de k termos $n \cdot n \ldots n = n_k$ como o número de cadeias de k elementos.

Permutações

Uma ***permutação*** de um conjunto finito S é uma sequência ordenada de todos os elementos de S, sendo que cada elemento aparece exatamente uma vez. Por exemplo, se $S = \{a, b, c\}$, então S tem seis permutações:

$$abc, acb, bac, bca, cab, cba.$$

Há $n!$ permutações de um conjunto de n elementos, visto que o primeiro elemento da sequência pode ser escolhido de n modos, o segundo de $n - 1$ modos, o terceiro de $n - 2$ modos, e assim por diante.

Uma ***k-permutação*** de S é uma sequência ordenada de k elementos de S na qual nenhum elemento aparece mais de uma vez (assim, uma permutação comum é apenas uma permutação de n elementos de um conjunto de n elementos). As doze 2-permutações do conjunto $\{a, b, c, d\}$ são

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc.$$

O número de k-permutações de um conjunto de n elementos é

$$n(n-1)(n-2) \ldots (n-k+1) = \frac{n!}{(n-k)!}, \qquad \text{(C.1)}$$

visto que há n modos de escolher o primeiro elemento, $n - 1$ modos de escolher o segundo elemento, e assim por diante, até selecionarmos k elementos, sendo o último elemento uma seleção dos $n - k + 1$ elementos restantes.

Combinações

Uma ***k-combinação*** de um conjunto S de n elementos é simplesmente um k-subconjunto de S. Por exemplo, o conjunto de quatro elementos $\{a, b, c, d\}$ tem seis 2-combinações:

$$ab, ac, ad, bc, bd, cd .$$

(Aqui usamos a forma reduzida de denotar o conjunto de dois elementos $\{a, b\}$ por ab, e assim por diante.) Podemos construir uma k-combinação de um conjunto de n elementos escolhendo k elementos distintos (diferentes) do conjunto de n elementos.

Podemos expressar o número de combinações de k elementos de um conjunto de n elementos em termos do número de permutações de k elementos de um conjunto de n elementos. Toda k-combinação tem exatamente $k!$

permutações de seus elementos, cada uma das quais é uma k-permutação distinta do n-conjunto. Assim, o número de k-combinações de um n-conjunto é o número de k-permutações dividido por $k!$; pela equação (C.1), essa quantidade é

$$\frac{n!}{k!(n-k)!} \tag{C.2}$$

Para $k = 0$, essa fórmula nos informa que o número de modos de escolher 0 elementos de um n-conjunto é 1 (e não 0), já que $0! = 1$.

Coeficientes binomiais

A notação $\binom{n}{k}$ (lê-se "n escolhe k") denota o número de k-combinações em um n-conjunto. Pela equação (C.2), temos

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Essa fórmula é simétrica em k e em $n - k$:

$$\binom{n}{k} = \binom{n}{n-k} \tag{C.3}$$

Esses números também são conhecidos como ***coeficientes binomiais*** porque aparecem na expansão binomial:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k \, y^{n-k} \tag{C.4}$$

Um caso especial da expansão binomial ocorre quando $x = y = 1$:

$$2^n = \sum_{k=0}^{n} \binom{n}{k}$$

Essa fórmula corresponde a contar as 2_n cadeias binárias de n elementos pelo número de 1s que elas contêm: n-cadeias binárias $\binom{n}{k}$ contêm exatamente k 1s, já que há $\binom{n}{k}$ modos de escolher k dentre as n posições nas quais colocar os 1*s*.

Muitas identidades envolvem coeficientes binomiais. Os exercícios no final desta seção lhe dão a oportunidade de provar algumas delas.

Limites para binomiais

Às vezes, precisamos limitar o tamanho de um coeficiente binomial. Para $1 \le k \le n$, temos o limite inferior

$$\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1}$$
$$= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right)$$
$$\geq \left(\frac{n}{k}\right)^k .$$

Tirando proveito da desigualdade $k! \geq (k/e)^k$ deduzida da aproximação de Stirling (3.18), obtemos os limites superiores

$$\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1}$$
$$\leq \frac{n^k}{k!}$$
$$\leq \left(\frac{en}{k}\right)^k \tag{C.5}$$

Para todos os inteiros k tais que $0 \leq k \leq n$, podemos usar a indução (veja o Exercício C.1-12) para provar o limite

$$\binom{n}{k} \leq \frac{n^n}{k^k(n-k)^{n-k}}. \tag{C.6}$$

onde, por conveniência, adotamos $0^0 = 1$. Para $k = ln$, onde $0 \leq l \leq 1$, podemos reescrever esse limite como

$$\binom{n}{\lambda n} \leq \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}}$$
$$= \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n$$
$$= 2^{nH(\lambda)},$$

onde

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda) \tag{C.7}$$

é a ***função entropia*** (***binária***) e onde, por conveniência, adotamos $0 \lg 0 = 0$, de modo que $H(0) = H(1) = 0$.

Exercícios

C.1-1 Quantas subcadeias de k elementos tem uma cadeia de n elementos? (Considere subcadeias idênticas de k elementos em posições diferentes como subcadeias diferentes.) Quantas subcadeias uma cadeia de n elementos tem no total?

C.1-2 Uma ***função booleana*** de n entradas e m saídas é função de $\{\text{TRUE}, \text{FALSE}\}_n$ para $\{\text{TRUE}, \text{FALSE}\}^m$. Quantas funções booleanas de n entradas e 1 saída existem? Quantas funções booleanas de n entradas e m saídas

existem?

C.1-3 De quantos modos n professores podem se sentar em torno de uma mesa de reuniões redonda? Considere duas arrumações iguais se uma pode ser rodada para formar a outra.

C.1-4 De quantos modos podemos escolher três números distintos no conjunto {1, 2, ..., 99} de modo que sua soma seja par?

C.1-5 Prove a identidade

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \qquad \text{(C.8)}$$

para $0 < k \leq n$.

C.1-6 Prove a identidade

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

para $0 \leq k < n$.

C.1-7 Para escolher k objetos de n, você pode destacar um dos objetos e considerar se tal objeto diferenciado é escolhido. Use essa abordagem para provar que

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

C.1-8 Usando o resultado do Exercício C.1-7, organize uma tabela para n = 0, 1, ..., 6 e $0 \leq k \leq n$ dos coeficientes binomiais ($_n{}^k$) que tenham (^{00}) na parte superior e (^{10}) na linha seguinte, e assim por diante. Essa tabela de coeficientes binomiais é denominada ***tri-*** ângulo de Pascal.

C.1-9 Prove que

$$\sum_{i=1}^{n} i = \binom{n+1}{2}$$

C.1-10 Mostre que, para quaisquer inteiros $n \geq 0$ e $0 \leq k \leq n$, a expressão ($_n$) alcança seu valor máximo quando k = n/2 ou k = n/2.

C.1-11 ★ Demonstre que, para quaisquer inteiros $n \geq 0$, $j \geq 0$, $k \geq 0$ e $j + k \leq n$,

$$\binom{n}{j+k} \le \binom{n}{j}\binom{n-j}{k} \quad \text{(C.9)}$$

Dê uma prova algébrica e também uma demonstração baseadas em um método para escolher $j + k$ itens de n itens. Dê um exemplo no qual a igualdade não seja válida.

C.1-12 ★ Use indução para todos os inteiros k tais que $0 \le k \le n/2$ para provar a desigualdade (C.6), e use a equação (C.3) para estendê-la a todos os inteiros k tais que $0 \le k \le n$.

C.1-13 ★ Use a aproximação de Stirling para provar que

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}(1+O(1/n)) \quad \text{(C.10)}$$

C.1-14 ★ Diferenciando a função entropia $H(l)$, mostre que ela alcança seu valor máximo em $l = 1/2$. O que é $H(1/2)$?

C.1-15 ★ Mostre que, para qualquer inteiro $n \ge 0$,

$$\sum_{k=0}^{n}\binom{n}{k}k = n2^{n-1} \quad \text{(C.11)}$$

C.2 PROBABILIDADE

Probabilidade é uma ferramenta essencial para o projeto e a análise de algoritmos probabilísticos e aleatorizados. Esta seção faz uma revisão da teoria básica da probabilidade.

Definimos probabilidade em termos de um ***espaço amostral*** S, que é um conjunto cujos elementos são denominados ***eventos elementares***. Cada evento elementar pode ser visto como um resultado possível de um experimento. No caso do experimento de lançar duas moedas distinguíveis, no qual cada lançamento individual resulta em uma cara (H) ou uma coroa (T) podemos considerar como espaço amostral o conjunto de todas as cadeias possíveis de dois elementos em {H, T}:

S = {HH, HT, TH, TT}.

Um ***evento*** é um subconjunto[1] do espaço amostral S. Por exemplo, no experimento de lançar duas moedas, o evento de obter uma cara e uma coroa é {HT, TH}. O evento S é denominado ***evento certo***, e o evento f é denominado ***evento nulo***. Dizemos que dois eventos A e B são ***mu- tuamente exclusivos*** se $A \cap B$ = /0. Algumas vezes tratamos um evento elementar $s \in S$ como o evento $\{s\}$. Por definição, todos os eventos elementares são mutuamente exclusivos.

Axiomas de probabilidade

Uma ***distribuição de probabilidades*** Pr{} em um espaço amostral S é um mapeamento de eventos de S para números reais que satisfaça os seguintes ***axiomas de probabilidade***:

1. $\Pr\{A\} \ge 0$ para qualquer evento A.

2. $\Pr\{S\} = 1$.

3. $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$ para quaisquer dois eventos mutuamente exclusivos A e B. De modo mais geral, para qualquer sequência de eventos (finita ou infinita contável) A_1, A_2, ... que sejam mutuamente exclusivos aos pares,

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\}$$

Denominamos $\Pr\{A\}$ a ***probabilidade*** do evento A. Aqui observamos que o axioma 2 é um requisito de normalização: na realidade, não há nada de fundamental em escolher 1 como a probabilidade do evento certo, exceto o fato de ser natural e conveniente.

Diversos resultados decorrem imediatamente desses axiomas e da teoria básica dos conjuntos (veja a Seção B.1). O evento nulo tem probabilidade $\Pr\{/0\} = 0$. Se $A \subseteq B$, então $\Pr\{A\} \leq \Pr\{B\}$. Usando A para denotar o evento $S - A$ (o ***complemento*** de A), temos $\Pr\{A\} = 1 - \Pr\{A\}$. Para dois eventos quaisquer A e B,

$$\begin{aligned} \Pr\{A \cup B\} &= \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} && \text{(C.12)} \\ &\leq \Pr\{A\} + \Pr\{B\} . && \text{(C.13)} \end{aligned}$$

Em nosso exemplo do lançamento de moedas, suponha que cada um dos quatro eventos elementares tenha probabilidade 1/4. Então, a probabilidade de obter no mínimo uma cara é

$$\begin{aligned} \Pr\{\text{HH}, \text{HT}, \text{TH}\} &= \Pr\{\text{HH}\} + \Pr\{\text{HT}\} + \Pr\{\text{TH}\} \\ &= 3/4 . \end{aligned}$$

Alternativamente, visto que a probabilidade de obter estritamente menos de uma cara é $\Pr\{\text{TT}\} = 1/4$, a probabilidade de obter no mínimo uma cara é $1 - 1/4 = 3/4$.

Distribuições de probabilidades discretas

Uma distribuição de probabilidades é ***discreta*** se é definida em um espaço amostral finito ou infinito contável. Seja S o espaço amostral. Então, para qualquer evento A,

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\},$$

já que eventos elementares, especificamente aqueles em A, são mutuamente exclusivos. Se S é finito e todo evento elementar $s \in S$ tem probabilidade $\Pr\{s\} = 1/|S|$,

$$\Pr\{s\} = 1/|S|,$$

então temos a ***distribuição de probabilidades uniforme*** em S. Em tal caso, o experimento é frequentemente descrito como "escolher um elemento de S aleatoriamente".

Como exemplo, considere o processo de lançar uma ***moeda não viciada***, uma moeda para a qual a probabilidade de obter uma cara é igual à probabilidade de obter uma coroa, ou seja, 1/2. Se lançarmos a moeda n vezes, temos a distribuição de probabilidades uniforme definida no espaço amostral $S = \{\text{H}, \text{T}\}_n$, um conjunto de tamanho 2_n. Podemos representar cada evento elementar em S como uma cadeia de comprimento n em {H, T}, sendo que cada cadeia ocorre com a probabilidade $1/2_n$. O evento

$A = \{$ocorrem exatamente k caras e exatamente $n - k$ coroas$\}$

é um subconjunto de S de tamanho $|A| = (_n{}^k)$, já que $(_n{}^k)$ cadeias de comprimento n em $\{\text{H}, \text{T}\}$ contêm exatamente k H's. Portanto, a probabilidade do evento A é $\Pr\{A\} = (_n{}^k)/2_n$.

Distribuição de probabilidade uniforme contínua

A distribuição de probabilidade uniforme contínua é um exemplo de distribuição de probabilidade na qual nem todos os subconjuntos do espaço amostral são considerados eventos. A distribuição de probabilidade uniforme contínua é definida em um intervalo fechado $[a, b]$ dos números reais, onde $a < b$. Nossa intuição é que cada ponto no intervalo $[a, b]$ deve ser "igualmente provável". Porém, há um número incontável de pontos; portanto, se dermos a todos os pontos a mesma probabilidade finita, positiva, não poderemos satisfazer simultaneamente os axiomas 2 e 3. Por essa razão, gostaríamos de associar uma probabilidade a somente *alguns* dos subconjuntos de S, de modo tal que os axiomas sejam satisfeitos para esses eventos.

Para qualquer intervalo fechado $[c, d]$, onde a c d b, a ***distribuição de probabilidade uniforme contínua*** define a probabilidade do evento $[c, d]$ como

$$\Pr\{c \mid d\} = \frac{d-c}{b-a} .$$

Observe que, para qualquer ponto $x = [x, x]$ a probabilidade de x é 0. Se eliminarmos as extremidades de um intervalo $[c, d]$, obteremos o intervalo aberto (c, d). Visto que $[c, d] = [c, c] \cup (c, \mathrm{d}) \cup [d, d]$, o axioma 3 nos dá $\Pr\{[c, d]\} = \Pr\{(c, d)\}$. Em geral, o conjunto de eventos para a distribuição de probabilidade uniforme contínua contém qualquer subconjunto do espaço amostral $[a, b]$ que possa ser obtido por uma união finita ou contável de intervalos abertos e fechados, bem como certos conjuntos mais complicados.

Probabilidade condicional e independência

Às vezes, temos algum conhecimento parcial antecipado sobre o resultado de um experimento. Por exemplo, suponha que um amigo tenha lançado duas moedas não viciadas e lhe tenha dito que no mínimo uma das moedas deu cara. Qual é a probabilidade de ambas as moedas darem caras? A informação dada elimina a possibilidade de duas coroas. Os três eventos elementares restantes são igualmente prováveis, portanto inferimos que cada um ocorre com probabilidade 1/3. Visto que apenas um desses eventos elementares dá duas caras, a resposta à nossa pergunta é 1/3.

A probabilidade condicional formaliza a noção de existir um conhecimento parcial antecipado do resultado de um experimento. A ***probabilidade condicional*** de um evento A dada a ocorrência de um outro evento B é definida como

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \qquad \text{(C.14)}$$

sempre que $\Pr\{B\} \neq 0$. (Lê-se "$\Pr\{A \mid B\}$" como "a probabilidade de A dado B".) Intuitivamente, já que sabemos que o evento B ocorre, a probabilidade de o evento A também ocorrer é $A \cap B$. Isto é, $A \cap B$ é o conjunto de resultados em que ocorrem ambos, A e B. Visto que o resultado é um dos eventos elementares em B, normalizamos as probabilidades de todos os eventos elementares em B dividindo-as por $\Pr\{B\}$, de tal forma que sua soma seja 1. Portanto, a probabilidade condicional de A dado B é a razão entre a probabilidade do evento $A \cap B$ e a probabilidade do evento B. Em nosso exemplo, A é o evento em que ambas as moedas dão caras e B é o evento em que no mínimo uma moeda dá cara. Assim, $\Pr\{A|B\} = (1/4)/(3/4) = 1/3$.

Dois eventos são ***independentes*** se

$$\Pr\{A \cap B\} = \Pr\{A\}\Pr\{B\} , \qquad \text{(C.15)}$$

que é equivalente, se $\Pr\{B\} \neq 0$, à condição

$$\Pr\{A|B\} = \Pr\{A\} .$$

Por exemplo, suponha que lancemos duas moedas não viciadas e que os resultados sejam independentes. Então, a probabilidade de duas caras é (1/2)(1/2) = 1/4. Agora, suponha que um evento seja a primeira moeda dar cara e o outro evento seja as moedas darem resultados diferentes. Cada um desses eventos ocorre com probabilidade 1/2, e a probabilidade de que ambos os eventos ocorram é 1/4; assim, de acordo com a definição de independência, os eventos são independentes — ainda que possamos imaginar que ambos os eventos dependem da primeira moeda. Finalmente, suponha que as moedas estejam soldadas de tal forma que ambas dão cara ou ambas dão coroa e que as duas possibilidades sejam igualmente prováveis. Então, a probabilidade de cada moeda dar cara é 1/2, mas a probabilidade de ambas darem cara é $1/2 \neq (1/2)(1/2)$. Por consequência, o evento em que uma das moedas dá cara e o evento em que a outra dá cara não são independentes.

Dizemos que uma coleção $A_1, A_2, ..., A_n$ de eventos é ***independente aos pares*** se

$$Pr\{A_i \cap A_j\} = Pr\{A_i\}Pr\{A_j\}$$

para todo $1 \leq i < j \leq n$. Dizemos que os eventos da coleção são (***mutuamente***) ***independentes*** se todo subconjunto de k elementos da coleção $A_{i_1}, A_{i_2}, ..., A_{i_k}$ em que $2 \leq k \leq n$ e $1 \leq i_1 < i_2 < ... < i_k \leq n$, satisfaz

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\} \cdots \Pr\{A_{i_k}\}$$

Por exemplo, suponha que lancemos duas moedas não viciadas. Seja A_1 o evento em que a primeira moeda dá cara, seja A_2 o evento em que a segunda moeda dá cara e seja A_3 o evento em que as duas moedas dão resultados diferentes. Temos

$$\begin{aligned}
\Pr\{A_1\} &= 1/2 \,, \\
\Pr\{A_2\} &= 1/2 \,, \\
\Pr\{A_3\} &= 1/2 \,, \\
\Pr\{A_1 \cap A_2\} &= 1/4 \,, \\
\Pr\{A_1 \cap A_3\} &= 1/4 \,, \\
\Pr\{A_2 \cap A_3\} &= 1/4 \,, \\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0 \,.
\end{aligned}$$

Visto que, para $1 \leq i < j \leq 3$, temos $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$, os eventos A_1, A_2 e A_3 são independentes aos pares. Contudo, os eventos não são mutuamente independentes porque $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$ e $\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\} = 1/8 \neq 0$.

Teorema de Bayes

Pela definição de probabilidade condicional (C.14) e da lei comutativa $A \cap B = B \cap A$, decorre que, para dois eventos A e B, cada um com probabilidade não nula,

$$\begin{aligned}
\Pr\{A \cap B\} &= \Pr\{B\}\Pr\{A|B\} \\
&= \Pr\{A\}\Pr\{B|A\} \,.
\end{aligned} \tag{C.16}$$

Resolvendo para $\Pr\{A|B\}$, obtemos

$$\Pr\{A \mid B\} = \frac{\Pr\{B\}\Pr\{B \mid A\}}{\Pr\{B\}}, \tag{C.17}$$

que é conhecido ***teorema de Bayes***. O denominador $\Pr\{B\}$ é uma constante de normalização que podemos expressar novamente da seguinte maneira. Considerando que $B = (B \cap A) \cup (B \cap A)$ e que $B \cap A$ e $B \cap A$ são eventos mutuamente exclusivos,

$$\begin{aligned}\Pr\{B\} &= \Pr\{B \cap \mathrm{A}\} + \Pr\{B \cap \overline{A}\} \\ &= \Pr\{A\} \Pr\{B \mid A\} + \Pr\{\overline{A}\} \Pr\{B|\overline{A}\} .\end{aligned}$$

Substituindo na equação (C.17), obtemos uma forma equivalente do teorema de Bayes:

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\{\bar{A}\}\Pr\{B \mid \bar{A}\}} \tag{C.18}$$

O teorema de Bayes pode simplificar o cálculo de probabilidades condicionais. Por exemplo, suponha que tenhamos uma moeda não viciada e uma moeda viciada que sempre dá cara. Executamos um experimento que consiste em três eventos independentes: escolhemos uma das duas moedas aleatoriamente, lançamos essa moeda uma vez e depois a lançamos mais uma vez. Suponha que a moeda escolhida dê cara ambas as vezes. Qual é a probabilidade de ela ser viciada?

Resolvemos esse problema usando o teorema de Bayes. Seja A o evento em que escolhemos a moeda viciada e seja B o evento em que a moeda dá cara ambas as vezes. Desejamos determinar $Pr\{A|B\}$. Temos $Pr\{A\} = 1/2$, $Pr\{B|A\} = 1$, $Pr\{A\} = 1/2$ e $\Pr\{B|A\} = 1/4$; consequentemente,

$$\begin{aligned}\Pr\{A \mid B\} &= \frac{(1/2)\cdot 1}{(1/2)\cdot 1 + (1/2)\cdot(1/4)} \\ &= 4/5.\end{aligned}$$

Exercícios

C.2-1 O professor Rosencrantz lança uma moeda não viciada uma vez. O professor Guildenstern lança uma moeda não viciada duas vezes. Qual é a probabilidade de o professor Rosencrantz obter mais caras que o professor Guildenstern?

C.2-2 Prove a ***desigualdade de Boole***: para qualquer sequência finita ou infinita contável de eventos $A_1, A_2, \ldots$,

$$\Pr\{A_1 \cup A_2 \cup \ldots\} \leq \Pr\{A_1\} + \Pr\{A_2\} + \ldots . \tag{C.19}$$

C.2-3 Suponha que embaralhemos muito bem um baralho de 10 cartas e que cada uma das cartas represente um número distinto de 1 até 10. Então, retiramos três cartas do baralho, uma de cada vez. Qual é a probabilidade de selecionarmos as três cartas em sequência ordenada (crescente)?

C.2-4 Prove que

$$\Pr\{A|B\} + \Pr\{\overline{A}|B\} = 1.$$

C.2-5 Prove que, para qualquer coleção de eventos $A_1, A_2, \ldots, A_n$,

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \ldots \cap A_n\} &= \Pr(A_1\} \cdot \Pr\{A_2|A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \ldots \\ &= \Pr\{A_n|A_1 \cap A_2 \cap \ldots \cap A_{n-1}\} .\end{aligned}$$

C.2-6 ★ Descreva um procedimento que toma como entrada dois inteiros a e b tais que $0 < a < b$ e, usando lançamentos de uma moeda não viciada, produz como resultado caras com probabilidade a/b e coroas com probabilidade $(b - a)/b$. Dê um limite para o número esperado de lançamentos da moeda que deve ser O(1).

(*Sugestão:* Represente a/b em binário.)

C.2-7 Mostre como construir um conjunto de n eventos independentes aos pares mas tais que nenhum subconjunto de $k > 2$ elementos desse conjunto seja mutuamente independente.

C.2-8 ★ Dois eventos A e B são ***condicionalmente independentes***, dado C, se

Dê um exemplo simples mas não trivial de dois eventos que não sejam independentes, mas condicionalmente independentes dado um terceiro evento.

C.2-9 ★ Você participa de um programa no qual um prêmio está escondido atrás de uma de três cortinas. Você ganhará o prêmio se selecionar a cortina correta. Depois de escolher uma cortina, mas antes de a cortina ser erguida, o apresentador ergue uma das outras cortinas, sabendo de antemão que isso revelará um cenário vazio e pergunta se você gostaria de trocar sua seleção atual pela cortina restante. De que modo suas chances mudarão se você trocar a seleção? (Essa pergunta é o famoso ***problema de Monty Hall***, que deve seu nome ao apresentador do programa de prêmios que frequentemente propunha aos participantes exatamente esse dilema.)

C.2-10 ★ O diretor de um presídio escolheu aleatoriamente um de três prisioneiros para ser libertado. Os outros dois serão executados. O carcereiro sabe qual deles será libertado, mas é proibido de dar a qualquer prisioneiro informações relativas a seu *status*. Vamos denominar os prisioneiros X, Y e Z. O prisioneiro X pergunta reservadamente ao carcereiro qual dos outros prisioneiros, Y ou Z, será executado argumentando que, visto que ele já sabe que no mínimo um deles deve morrer, o carcereiro não estará revelando quaisquer informações sobre o seu próprio *status*. O carcereiro diz a X que Y deverá ser executado. O prisioneiro X se sente mais feliz agora, já que chegou à conclusão de que ele ou o prisioneiro Z será libertado, o que significa que agora sua probabilidade de ganhar a liberdade é 1/2. Ele está certo ou sua chance de viver ainda é de 1/3? Explique.

C.3 VARIÁVEIS ALEATÓRIAS DISCRETAS

Uma ***variável aleatória (discreta)*** X é uma função de um espaço amostral finito ou infinito contável S para os números reais. Ela associa um número real a cada resultado possível de um experimento, o que nos permite trabalhar com a distribuição de probabilidades induzida no conjunto de números resultante. Variáveis aleatórias também podem ser definidas para espaços amostrais infinitos incontáveis, mas isso dá origem a questões técnicas que não há necessidade de abordar, dadas as nossas finalidades. Daqui em diante, suporemos que variáveis aleatórias são discretas.

Para uma variável aleatória X e um número real x, definimos o evento $X = x$ como $\{s \in S : X(s) = x\}$; portanto,

$$\Pr\{X = x\} = \sum_{s \in S : X(s) = x} \Pr\{s\}$$

A função

$$f(x) = \Pr\{X = x\}$$

é a ***função densidade de probabilidade*** da variável aleatória X. Pelos axiomas de probabilidade, $\Pr\{X = x\} \geq 0$ e $\sum_x \Pr\{X = x\} = 1$.

Como exemplo, considere o experimento de lançar um par de dados comuns de seis faces. Há 36 eventos elementares possíveis no espaço amostral. Supomos que a distribuição de probabilidades é uniforme, de modo que cada evento elementar $s \in S$ é igualmente provável: $\Pr\{s\} = 1/36$. Defina a variável aleatória X como o *máximo* dos

dois valores resultantes do lançamento dos dados. Temos $Pr\{X = 3\} = 5/36$, já que X atribui um valor de 3 a 5 dos 36 eventos elementares possíveis, isto é, (1, 3), (2, 3), (3, 3), (3, 2) e (3, 1).

Muitas vezes, definimos várias variáveis aleatórias no mesmo espaço amostral. Se X e Y são variáveis aleatórias, a função

$$f(x, y) = \Pr\{X = x \text{ e } Y = y\}$$

é a ***função densidade de probabilidade conjunta*** de X e Y. Para um valor fixo y,

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ e } Y = y\},$$

e, de modo semelhante, para um valor fixo x,

$$\Pr\{X = x\} = \sum_x \Pr\{X = x \text{ e } Y = y\}.$$

Usando a definição (C.14) de probabilidade condicional, temos

$$\Pr\{X = x \mid Y = y\} = \frac{\Pr\{X = x \text{ e } Y = y\}}{\Pr\{Y = y\}}.$$

Dizemos que duas variáveis aleatórias X e Y são ***independentes*** se, para todo x e y, os eventos $X = x$ e $Y = y$ são independentes ou, de modo equivalente, se para todo x e y, temos $\Pr\{X = x \text{ e } Y = y\} = \Pr\{X = x\}\Pr\{Y = y\}$.

Dado um conjunto de variáveis aleatórias definidas em um mesmo espaço amostral, podemos definir novas variáveis aleatórias como somas, produtos ou outras funções das variáveis originais.

Valor esperado de uma variável aleatória

O resumo mais simples e mais útil da distribuição de uma variável aleatória é a "média" dos valores que ela adota. O ***valor esperado*** (ou os sinônimos ***esperança*** ou ***média***) de uma variável aleatória discreta X é

$$E[X] = \sum_x x \cdot \Pr\{X = x\}, \tag{C.20}$$

que é bem definido se a soma é finita ou absolutamente convergente. Às vezes, a esperança de X é denotada por m^x ou, quando a variável aleatória é aparente pelo contexto, simplesmente por m.

Considere um jogo em que você lança duas moedas não viciadas. Você ganha R$3,00 para cada cara, mas perde R$2,00 para cada coroa. O valor esperado da variável aleatória X que representa seus ganhos é

$$\begin{aligned} E[X] &= 6 \cdot \Pr\{2 \text{ H's}\} + 1 \cdot P\{1H, 1T\} - 4 \cdot \Pr\{2 \text{ T's}\} \\ &= 6(1/4) + 1(1/2) - 4(1/4) \\ &= 1\,. \end{aligned}$$

A esperança da soma de duas variáveis aleatórias é a soma de suas esperanças, isto é,

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y], \tag{C.21}$$

sempre que E[X] e E[Y] são definidos. Denominamos essa propriedade ***linearidade da esperança***, e ela é válida até mesmo se X e Y não são independentes. Ela também se estende a somatórios de esperanças, tanto finitos quanto absolutamente convergentes. A linearidade de esperança é a propriedade fundamental que nos permite executar análises probabilísticas utilizando variáveis aleatórias indicadoras (consulte a Seção 5.2).

Se X é qualquer variável aleatória, qualquer função $g(x)$ define uma nova variável aleatória $g(X)$. Se a esperança de $g(X)$ é definida, então

$$E\left[g(X)\right] = \sum_x g(x) \cdot \Pr\{X = x\}$$

Fazendo $g(x) = ax$ temos, para qualquer constante a,

$$\mathrm{E}[aX] = a\mathrm{E}[X] \,. \tag{C.22}$$

Consequentemente, esperanças são lineares: para quaisquer duas variáveis aleatórias X e Y e uma constante qualquer a,

$$\mathrm{E}[aX + Y] = a\mathrm{E}[X] + \mathrm{E}[Y] \,. \tag{C.23}$$

Quando duas variáveis aleatórias X e Y são independentes e cada uma tem uma esperança definida,

$$\begin{aligned}
\mathrm{E}\{X, Y\} &= \sum_x \sum_y xy \cdot \Pr\{X = x \text{ e } Y = y\} \\
&= \sum_x \sum_y xy \cdot \Pr\{X = x\} \Pr\{Y = y\} \\
&= \left(\sum\nolimits_x x \cdot \Pr\{X = x\}\right)\left(\sum\nolimits_y y \cdot \Pr\{Y = y\}\right) \\
&= \mathrm{E}[X]\mathrm{E}[Y]
\end{aligned}$$

Em geral, quando n variáveis aleatórias $X_1, X_2, \ldots, X_n$ são mutuamente independentes,

$$\mathrm{E}[X_1 X_2 \cdots X_n] = \mathrm{E}[X_1]\, E[X_2] \cdots \mathrm{E}[X_n] \,. \tag{C.24}$$

Quando uma variável aleatória X assume valores pertencentes ao conjunto dos números naturais $= \{0, 1, 2, \ldots\}$, existe uma fórmula elegante para representar sua esperança:

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=0}^{\infty} i \cdot \Pr\{X = i\} \\
&= \sum_{i=0}^{\infty} i \cdot (\Pr\{X \geq i\} - \Pr\{X \geq i + 1\}) \\
&= \sum_{i=0}^{\infty} \Pr\{X \geq i\} \,,
\end{aligned} \tag{C.25}$$

visto que cada termo $\Pr\{X \geq i\}$ é somado i vezes e subtraído em $i - 1$ vezes (exceto $\Pr\{X \geq 0\}$, que é somado 0 vezes e não é subtraído).

Quando aplicamos uma função convexa $f(x)$ a uma variável aleatória X, a ***desigualdade de Jensen*** nos dá

$$\mathrm{E}[f(X)] \geq f(\mathrm{E}[X]) \,, \tag{C.26}$$

desde que as esperanças existam e sejam finitas. (Uma função $f(x)$ é ***convexa*** se para todo x e y, e para todo $0 \leq l \leq 1$, temos $f(lx + (1 - l)y) \leq l\, f(x) + (1 - l) f(y)$.)

Variância e desvio-padrão

O valor esperado de uma variável aleatória não nos informa como os valores da variável estão "espalhados". Por exemplo, se temos variáveis aleatórias X e Y para as quais $\Pr\{X = 1/4\} = \Pr\{X = 3/4\} = 1/2$ e $\Pr\{Y = 0\} = \Pr\{Y = 1\} = 1/2$, então tanto $E\{X\}$ quanto $E\{Y\}$ são 1/2, ainda que os valores reais adotados por Y estejam mais distantes da média que os valores reais adotados por X.

A noção de variância expressa matematicamente quão afastados da média os valores de uma variável aleatória provavelmente estão. A ***variância*** de uma variável aleatória X com média $\mathrm{E}[X]$ é

$$
\begin{aligned}
\text{Var}[X] &= \text{E}[(X - \text{E}[X])^2] \\
&= \text{E}[X^2 - 2X\text{E}[X] + \text{E}^2[X]] \\
&= \text{E}[X^2] - 2\text{E}[X\text{E}][X]] + \text{E}^2[X] \\
&= \text{E}[X^2] - 2\text{E}^2[X] + \text{E}^2[X] \\
&= \text{E}[X^2] - \text{E}^2[X] \,. \qquad \text{(C.27)}
\end{aligned}
$$

Para justificar a igualdade $\text{E}[\text{E}^2[X$ $= \text{E}^2[X]$, observe que, como $\text{E}[X]$ é um número real e não uma variável aleatória, $\text{E}[\text{E}^2[X$ também é um número real. A igualdade $\text{E}[X\text{E}[X$ $= \text{E}^2[X]$ decorre da equação (C.22), com $a = \text{E}[X]$). Reescrevendo a equação (C.27) obtemos uma expressão para a esperança do quadrado de uma variável aleatória:

$$\text{E}[X^2] = \text{Var}[X] + \text{E}^2[X]. \qquad \text{(C.28)}$$

A variância de uma variável aleatória X e a variância de aX estão relacionadas (veja o Exercício C.3-10):

$$\text{Var}[aX] = a^2\text{Var}[X] \,.$$

Quando X e Y são variáveis aleatórias independentes,

$$\text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y] \,.$$

Em geral, se n variáveis aleatórias $X_1, X_2, ..., X_n$ são independentes aos pares, então

$$Var\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} Var[X_i] \qquad \text{(C.29)}$$

O ***desvio-padrão*** de uma variável aleatória X é a raiz quadrada positiva da variância de X. O desvio-padrão de uma variável aleatória X, às vezes, é denotado por $_x$, ou simplesmente , quando a variável aleatória X é entendida pelo contexto. Com essa notação, a variância de X é denotada por ^{2}.

Exercícios

C.3-1 Suponha que lancemos dois dados comuns de seis faces. Qual é a esperança da soma dos dois valores exibidos resultantes? Qual é a esperança do máximo dos dois valores resultantes?

C.3-2 Um arranjo $A[1..n]$ contém n números distintos que estão ordenados aleatoriamente, sendo que cada permutação dos n números é igualmente provável. Qual é a esperança do índice do elemento máximo no arranjo? Qual é a esperança do índice do elemento mínimo no arranjo?

C.3-3 Um certo jogo de parque de diversões consiste em três dados dentro de uma gaiola giratória. Um jogador pode apostar R$1,00 em qualquer dos números de 1 a 6. A gaiola é girada e o resultado é o seguinte: se o número escolhido pelo jogador não aparecer em nenhum dos dados, ele perde seu dinheiro. Caso contrário, se seu número aparecer em exatamente k dos três dados, para $k = 1, 2, 3$, o jogador mantém seu dinheiro e ganha k vezes o valor apostado. Qual é o ganho esperado quando se aposta nesse jogo uma vez?

C.3-4 Demonstre que, se X e Y são variáveis aleatórias não negativas, então

$$\text{E}[\max(X, Y)] \leq \text{E}[X] + \text{E}[Y] \,.$$

C.3-5 ★ Sejam X e Y variáveis aleatórias independentes. Prove que $f(X)$ e $g(Y)$ são independentes para qualquer escolha de funções f e g.

C.3-6 ★ Seja X uma variável aleatória não negativa e suponha que E[X] esteja bem definida. Prove a ***desigualdade de Markov***:

$$\Pr\{X \geq t\} \leq E[X]/t \qquad \text{(C.30)}$$

para todo $t > 0$.

C.3-7 ★ Seja S um espaço amostral e sejam X e X' variáveis aleatórias tais que $X(s) \geq X'(s)$ para todo $s \in S$. Prove que, para qualquer constante real t,

$$\Pr\{X \geq t\} \geq \Pr[X' \geq t]$$

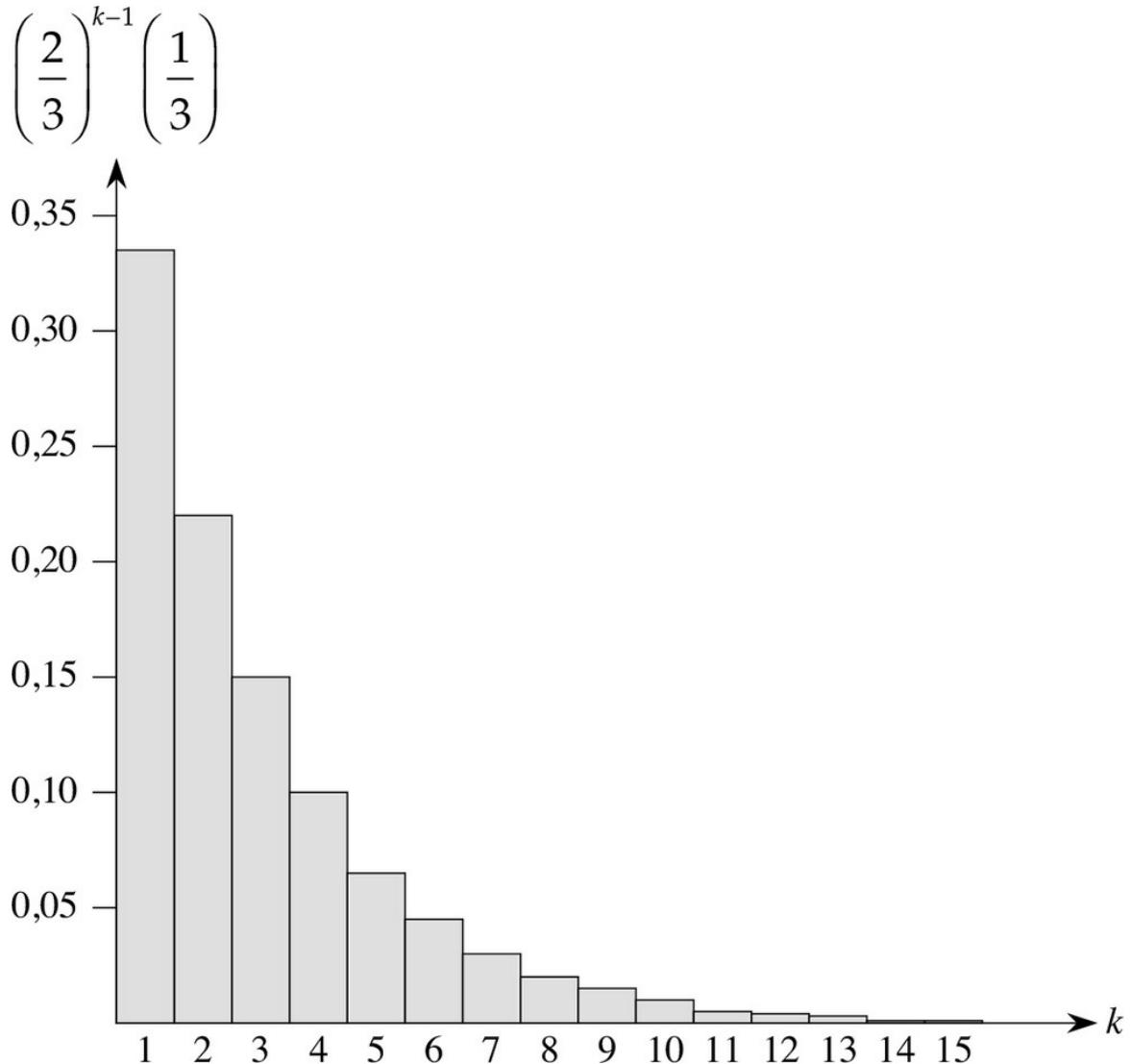
C.3-8 O que é maior: a expectativa do quadrado de uma variável aleatória ou o quadrado de sua esperança?

C.3-9 Mostre que, para qualquer variável aleatória X que adote somente os valores 0 e 1, temos Var[X] = $E[X]E[1 - X]$.

C.3-10 Prove que Var[aX] = a_2Var[X], pela definição (C.27) de variância.

C.4 DISTRIBUIÇÕES GEOMÉTRICA E BINOMIAL

Podemos pensar em um lançamento de moeda como uma instância de uma ***tentativa de Bernoulli***, que é um experimento que tem somente dois resultados possíveis: ***sucesso***, que ocorre com probabilidade p, e ***insucesso***, que ocorre com probabilidade $q = 1 - p$. Quando falamos coletivamente de ***tentativas de Bernoulli***, queremos dizer que as tentativas são mutuamente independentes e, a menos que digamos especificamente o contrário, cada uma delas tem a mesma probabilidade p de sucesso. Duas distribuições importantes surgem das tentativas de Bernoulli: a distribuição geométrica e a distribuição binomial.

A distribuição geométrica

Suponha que tenhamos uma sequência de tentativas de Bernoulli, cada uma com uma probabilidade de sucesso p e uma probabilidade $q = 1 - p$ de insucesso. Quantas tentativas ocorrem antes de obtermos um sucesso? Vamos definir a variável aleatória X como o número de tentativas necessárias para obter um sucesso. Então, X tem valores no intervalo {1, 2, ...} e, para $k \geq 1$,

$$\Pr\{X = k\} = q^{k-1} p, \qquad \text{(C.31)}$$

já que temos $k - 1$ insucessos antes de um sucesso. Dizemos que uma distribuição de probabilidades que satisfaz a equação (C.31) é uma ***distribuição geométrica***. A Figura C.1 ilustra tal distribuição.

Considerando que $p < 1$, podemos calcular a esperança de uma distribuição geométrica utilizando a identidade (A.8):

$$\begin{aligned}E[X] &= \sum_{k=1}^{\infty} kq^{k-1}p \\ &= \frac{p}{q}\sum_{k=0}^{\infty} kq^{k} \\ &= \frac{p}{q}\cdot\frac{q}{(1-q)^2} \\ &= \frac{p}{q}\cdot\frac{q}{p^2} \\ &= 1/p\end{aligned} \tag{C.32}$$

Figura C.1 Uma distribuição geométrica com probabilidade de sucesso $p = 1/3$ e uma probabilidade de insucesso $q = 1 - p$. A esperança da distribuição é $1/p = 3$.

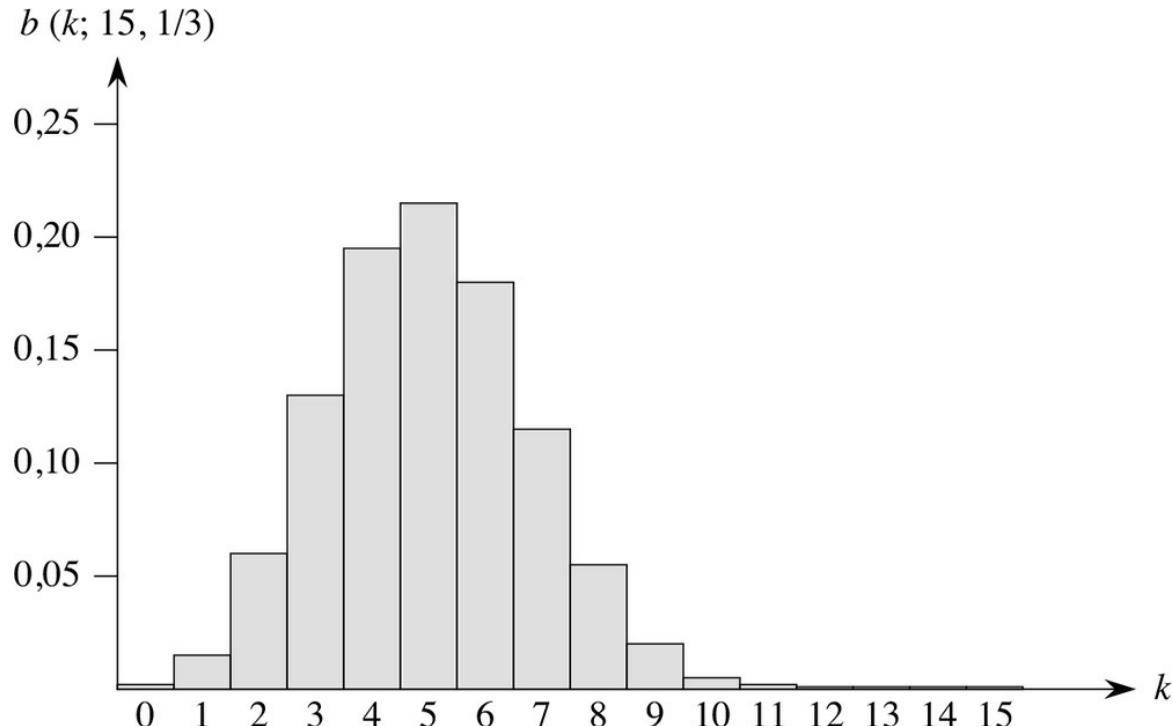
Assim, em média, ocorrem $1/p$ tentativas antes de obtermos um sucesso, um resultado intuitivo. A variância, que pode ser calculada de modo semelhante, mas usando o Exercício A.1-3, é

$$Var[X] = q/p^2\ . \tag{C.33}$$

Como exemplo, suponha que lancemos dois dados repetidas vezes até obtermos um sete ou um onze. Dos 36 resultados possíveis, seis produzem um sete e dois produzem um onze. Assim, a probabilidade de sucesso é $p = 8/36 = 2/9$, e temos de lançar os dados $1/p = 9/2 = 4{,}5$ vezes em média para obter um valor sete ou onze.

A distribuição binomial

Quantos sucessos ocorrem durante n tentativas de Bernoulli, onde um sucesso ocorre com probabilidade p e um insucesso com probabilidade $q = 1 - p$? Defina a variável aleatória X como o número de sucessos em n tentativas. Então, X tem valores na faixa $\{0, 1, ..., n\}$ e, para $k = 0, 1,..., n$,

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}, \quad \text{(C.34)}$$

visto que há $\binom{}{n}$ modos de escolher quais k das n tentativas são sucessos, e a probabilidade de ocorrer cada uma é $p_k q_n{}^{-k}$. Uma distribuição de probabilidades que satisfaz a equação (C.33) é denominada ***distribuição binomial***. Por conveniência, definimos a família de distribuições binomiais usando a notação

$$b(k; n, p) = \binom{n}{k} p^k (1 - p)^{n-k}. \quad \text{(C.35)}$$

A Figura C.2 ilustra uma distribuição binomial. O nome "binomial" deve-se ao fato de que o lado direito da equação (C.34) é o k-ésimo termo da expansão de $(p + q)_n$. Consequentemente, visto que $p + q = 1$,

$$\sum_{k=0}^{n} (k; n, p) - 1, \quad \text{(C.36)}$$

como é exigido pelo axioma 2 dos axiomas de probabilidade.

Figura C.2 A distribuição binomial de $b(k; 15, 1/3)$ resultante de $n = 15$ tentativas de Bernoulli, cada uma com probabilidade de sucesso $p = 1/3$. A expectativa da distribuição é $np = 5$.

Podemos calcular a esperança de uma variável aleatória cuja distribuição é binomial pelas equações (C.8) e (C.36). Seja X uma variável aleatória que segue a distribuição binomial $b(k; n, p)$, e seja $q = 1 - p$. Pela definição de esperança, temos

$$\begin{aligned}
E[X] &= \sum_{k=0}^{n} k \cdot \Pr\{X = k\} \\
&= \sum_{k=0}^{n} k \cdot b(k; n, p) \\
&= \sum_{k=0}^{n} k \binom{n}{k} p^k q^{n-k} \\
&= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} q^{n-k} && \text{(pela equação (C.8))} \\
&= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \\
&= np \sum_{k=0}^{n-1} b(k; n-1, p) \\
&= np && \text{(pela equação (C.36))} \qquad \text{(C.37)}
\end{aligned}$$

Usando a linearidade de esperança podemos obter o mesmo resultado com uma quantidade substancialmente menor de cálculos algébricos. Seja X_i a variável aleatória que descreve o número de sucessos na i-ésima tentativa. Então, $E[X_i] = p \cdot 1 + q \cdot 0 = p$ e, por linearidade de esperança (equação (C.21)), o número esperado de sucessos para n tentativas é

$$\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} E[X_i] \\
&= \sum_{i=1}^{n} p \\
&= np \qquad \text{(C.38)}
\end{aligned}$$

Podemos usar a mesma abordagem para calcular a variância da distribuição. Empregando a equação (C.27), temos $\text{Var}[X_i] = E[X_i^2] - E^2[X_i]$. Visto que X_i adota somente os valores 0 e 1, temos $X_i^2 = X_i$, o que implica $E\left[X_i^2\right] = E\left[X_i\right] = p$. Consequentemente,

$$\text{Var}[X_i] = p - p^2 = p(1 - p) = q. \qquad \text{(C.39)}$$

Para calcular a variância de X, aproveitamos a independência das n tentativas; assim, pela equação (C.29),

$$\begin{aligned}
\text{Var}[X] &= \text{Var}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \text{Var}[X_i] \\
&= \sum_{i=1}^{n} pq \\
&= npq\ . \qquad \text{(C.40)}
\end{aligned}$$

Como mostra a Figura C.2, a distribuição binomial $b(k; n, p)$ aumenta com k até alcançar a média np e depois diminui. Podemos provar que a distribuição sempre se comporta dessa maneira examinando a razão entre termos consecutivos:

$$\begin{aligned}
\frac{b(k;n,p)}{b(k-1;n,p)} &= \frac{\binom{n}{k}p^k q^{n-k}}{\binom{n}{k-1}p^{k-1}q^{n-k+1}} \\
&= \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \\
&= \frac{(n-k+1)p}{kq} \\
&= 1+\frac{(n+1)p-k}{kq} .
\end{aligned} \tag{C.41}$$

Essa razão é maior que 1 exatamente quando $(n + 1)p - k$ é positiva. Consequentemente, $b(k; n, p) > b(k - 1; n, p)$ para $k < (n + 1)p$ (a distribuição aumenta) e $b(k; n, p) < b(k - 1; n, p)$ para $k > (n + 1)p$ (a distribuição diminui). Se $k = (n + 1)p$ é um inteiro, então $b(k; n, p) = b(k - 1; n, p)$, e portanto a distribuição tem dois máximos: em $k = (n + 1)p$ e em $k - 1 = (n + 1)p - 1 = np - q$. Caso contrário, ela atinge um máximo no único inteiro k que se encontra na faixa $np - q < k < (n + 1)p$.

O lema a seguir dá um limite superior para a distribuição binomial.

Lema C.1

Seja $n \geq 0$, seja $0 < p < 1$, seja $q = 1 - p$ e seja $0 \leq k \leq n$. Então,

$$b(k;n,p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}$$

Prova Usando a equação (C.6), temos

$$\begin{aligned}
b(k;n,p) &= \binom{n}{k} p^k q^{n-k} \\
&\leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \\
&= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}
\end{aligned}$$

Exercícios

C.4-1 Verifique o axioma 2 dos axiomas da probabilidade para a distribuição geométrica.

C.4-2 Quantas vezes em média devemos lançar seis moedas não viciadas antes de obtermos três caras e três coroas?

C.4-3 Mostre que $b(k; n, p) = b(n - k; n, q)$, onde $q = 1 - p$.

C.4-4 Mostre que o valor do máximo da distribuição binomial $b(k; n, p)$ é aproximadamente $1/\sqrt{2npq}$, onde $q = 1 - p$.

C.4-5 ★ Mostre que a probabilidade de nenhum sucesso em n tentativas de Bernoulli, cada uma com probabilidade $p = 1/n$, é aproximadamente $1/e$. Mostre que a probabilidade de exatamente um sucesso é também aproximadamente $1/e$.

C.4-6 ★ O professor Rosencrantz lança uma moeda não viciada n vezes, e o professor Guildenstern faz o mesmo. Mostre que a probabilidade de eles obterem o mesmo número de caras é $\binom{2n}{n} / 4^n$. (*Sugestão*: Para o professor Rosencrantz, considere uma cara um sucesso; para o professor Guildenstern, considere uma coroa um sucesso.) Use sua demonstração para verificar a identidade

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}$$

C.4-7 Mostre que, para $0 \le k \le n$,

$$b(k; n, 1/2) \le 2^{n\,H(k/n) - n} ,$$

onde $H(x)$ é a função entropia (C.7).

C.4-8 Considere n tentativas de Bernoulli onde, para $i = 1, 2, ..., n$, a i-ésima tentativa tem uma probabilidade p_i de sucesso, e seja X a variável aleatória que denota o número total de sucessos. Seja $p \ge p_i$ para todo $i = 1, 2, ..., n$. Prove que, para $1 \le k \le n$,

$$\Pr\{X < k\} \ge \sum_{i=0}^{k-1} b(i; n, p)$$

C.4-9 ★ Seja X a variável aleatória para o número total de sucessos em um conjunto A de n tentativas de Bernoulli, onde a i-ésima tentativa tem uma probabilidade p_i de sucesso e seja X' a variável aleatória para o número total de sucessos em um segundo conjunto A' de n tentativas de Bernoulli, onde a i-ésima tentativa tem uma probabilidade de sucesso $p'_i \ge p_i$. Prove que, para $0 \le k \le n$,

$\Pr\{X' \ge k\} \ge \Pr\{X \ge k\}$.

(*Sugestão*: Mostre como obter as tentativas de Bernoulli em A' por meio de um experimento envolvendo as tentativas de A e use o resultado do Exercício C.3-7.)

C.5 AS CAUDAS DA DISTRIBUIÇÃO BINOMIAL

A probabilidade de haver no mínimo ou no máximo k sucessos em n tentativas de Bernoulli, cada um com probabilidade de sucesso p, é frequentemente de maior interesse do que a probabilidade de haver exatamente k sucessos. Nesta seção, investigamos as ***caudas*** da distribuição binomial: as duas regiões da distribuição $b(k; n, p)$ que estão longe do np médio. Demonstraremos vários limites importantes para (a soma de todos os termos de) uma cauda.

Primeiro, damos um limite para a cauda direita da distribuição $b(k; n, p)$. Podemos determinar limites para a cauda esquerda invertendo os papéis de sucessos e insucessos.

Teorema C.2

Considere uma sequência de n tentativas de Bernoulli, onde o sucesso ocorre com probabilidade p. Seja X a variável aleatória que denota o número total de sucessos. Então, para $0 \le k \le n$, a probabilidade de no mínimo k sucessos é

$$\begin{aligned}\Pr\{X \ge k\} &= \sum_{i=k}^{n} b(i;n,p) \\ &\le \binom{n}{k} p^k\end{aligned}$$

Prova Para $S \subseteq \{1, 2, ..., n\}$, seja A_S o evento em que a i-ésima tentativa é um sucesso para todo $i \in S$. É claro que $\Pr\{A_S\} = p_k$ se $|S| = k$. Temos

$$\begin{aligned}\Pr\{X \ge k\} &= \Pr\{\text{existe } S \subseteq \{1,2,...,n\} : |S| = k \text{ e } A_s\} \\ &= \Pr\left\{ \bigcup_{S\subseteq\{1,2,...,n\}:\,|S|=k} A_S \right\} \\ &\le \sum_{S\subseteq\{1,2,...,n\}:\,|S|=k} \Pr\{A_s\} \qquad \text{(pela desigualdade (C.19))} \\ &= \binom{n}{k} p^k.\end{aligned}$$

O corolário a seguir enuncia novamente o teorema para a cauda esquerda da distribuição binomial. Em geral, deixaremos a cargo do leitor adaptar as prova de uma cauda à outra.

Corolário C.3

Considere uma sequência de n tentativas de Bernoulli nas quais o sucesso ocorre com probabilidade p. Se X é a variável aleatória que denota o número total de sucessos, então para $0 \ \ k \ \ n$, a probabilidade de no máximo k sucessos é

$$\begin{aligned}\Pr\{X \le k\} &= \sum_{i=0}^{k} b(i;n,p) \\ &\le \binom{n}{n-k} (1-p)^{n-k} \\ &= \binom{n}{k} (1-p)^{n-k}.\end{aligned}$$

Nosso próximo limite refere-se à cauda esquerda da distribuição binomial. Seu corolário mostra que, longe da média, a cauda esquerda diminui exponencialmente.

Teorema C.4

Considere uma sequência de n tentativas de Bernoulli na qual o sucesso ocorre com probabilidade p e o insucesso com probabilidade $q = 1 - p$. Seja X a variável aleatória que denota o número total de sucessos. Então, para $0 < k < np$, a probabilidade de ocorrer um número menor do que k sucessos é

$$\begin{aligned}\Pr\{X < k\} &= \sum_{i=0}^{k-1} b(i;n,p) \\ &< \frac{kq}{np-k} b(k;n,p)\end{aligned}$$

Prova Limitamos a série $\sum_{i=0}^{k-1} b(i;n,p)$ por uma série geométrica utilizando a técnica da Seção A.2. Para $i = 1, 2, ..., k$ temos, pela equação (C.41)

$$\begin{aligned}\frac{b(i-1;n,p)}{b(i;n,p)} &= \frac{iq}{(n-i+1)p} \\ &< \frac{iq}{(n-i)p} \\ &\le \frac{kq}{(n-k)p}.\end{aligned}$$

Se fizermos

$$\begin{aligned}x &= \frac{kq}{(n-k)p} \\ &< \frac{kq}{(n-np)p} \\ &= \frac{kq}{nqp} \\ &= \frac{k}{np} \\ &< 1 .\end{aligned}$$

decorre que

$$b(i-1; n, p) < xb(i; n, p)$$

para $0 < i < k$. Aplicando iterativamente essa igualdade $k - i$ vezes, obtemos

$$b(i; n, p) < x^{k-i} b(k; n, p)$$

para $0 < i < k$ e, consequentemente,

$$
\begin{aligned}
\sum_{i=0}^{k-1} b(i;n,p) &< \sum_{i=0}^{k-1} x^{k-1} b(k;n,p) \\
&< b(k;n,p) \sum_{i=0}^{\infty} x^i \\
&= \frac{x}{1-x} b(k;n,p) \\
&= \frac{kq}{np-k} b(k;n,p)
\end{aligned}
$$

Corolário C.5

Considere uma sequência de n tentativas de Bernoulli, na qual o sucesso ocorra com probabilidade p e o insucesso com probabilidade $q = 1 - p$. Então, para $0 < k < np/2$, a probabilidade de um número de sucessos menor do que k é menor que metade da probabilidade de um número de sucessos menor do que $k + 1$.

Prova Como $k \leq np/2$, temos

$$
\begin{aligned}
\frac{kp}{np-k} &\leq \frac{(np/2)q}{np-(np/2)} \\
&= \frac{(np/2)q}{np/2} \\
&= 1 \,, \qquad \text{(C.42)}
\end{aligned}
$$

já que $q \leq 1$. Sendo X a variável aleatória que denota o número de sucessos, o Teorema C.4 e a desigualdade (C.42) implicam que a probabilidade de um número de sucessos menor do que k é

$$
\Pr\{X < k\} = \sum_{i=0}^{k-1} b(i;n,p) < b(k;n,p)
$$

Assim, temos

$$
\begin{aligned}
\frac{\Pr\{X < k\}}{\Pr\{X < k+1\}} &= \frac{\sum_{i=0}^{k-1} b(k;n,p)}{\sum_{i=0}^{k} b(i;n,p)} \\
&= \frac{\sum_{i=0}^{k-1} b(i;n,p)}{\sum_{i=0}^{k-1} b(i;n,p) + b(k;n,p)} \\
&< 1/2 \,,
\end{aligned}
$$

$$
\text{desde } \sum_{i=0}^{k-1} b(i;n;p) < b(k;n,p)
$$

Os limites para a cauda direita decorrem de modo semelhante. Você deve dar a prova no Exercício C.5-2.

Corolário C.6

Considere uma sequência de n tentativas de Bernoulli, na qual o sucesso ocorre com probabilidade p. Seja X a variável aleatória que denota o número total de sucessos. Então, para $np < k < n$, a probabilidade de um número de sucessos maior do que k é

$$\begin{aligned}\Pr\{X > k\} &= \sum_{i=k+1}^{n} b(i;n,p) \\ &< \frac{(n-k)p}{k-np} b(k;n,p)\end{aligned}$$

Corolário C.7

Considere uma sequência de n tentativas de Bernoulli na qual o sucesso ocorra com probabilidade p e o insucesso com probabilidade $q = 1 - p$. Então, para $(np + n)/2 < k < n$, a probabilidade de um número de sucessos maior do que k é menor que metade da probabilidade de um número de sucessos maior do que k - 1.

O próximo teorema considera n tentativas de Bernoulli cada uma com probabilidade p_i de sucesso, para $i = 1, 2, ..., n$. Como mostra o corolário subsequente, podemos usar o teorema para dar um limite para a cauda direita da distribuição binomial definindo $p_i = p$ para cada tentativa.

Teorema C.8

Considere uma sequência de n tentativas de Bernoulli na qual a i-ésima tentativa, para $i = 1, 2, ..., n$ ocorra sucesso com probabilidade p_i e o insucesso com probabilidade $q_i = 1 - p_i$. Seja X a variável aleatória que descreve o número total de sucessos, e seja m= E[X]. Então para $r > m$,

$$\Pr\{X - \mu \geq r\} \leq \left(\frac{\mu e}{r}\right)^r .$$

Prova Visto que, para qualquer $\alpha > 0$, a função e^{ax} é estritamente crescente em x,

$$\Pr\{X - \mu \geq r\} = \Pr\{e^{a(X-\mu)} \geq e^{ar}\}, \tag{C.43}$$

onde a será determinado mais adiante. Usando a desigualdade de Markov (C.30), obtemos

$$\Pr\{e^{a(X-\mu)} \geq e^{ar}\} \leq \mathrm{E}[e^{\alpha(X-m)}]\, e^{-ar}. \tag{C.44}$$

O grosso da prova consiste em limitar $\mathrm{E}[e_a(X - m)]$ e substituir a por um valor adequado na desigualdade (C.44). Primeiro, avaliamos $\mathrm{E}[e_a(X - m)]$. Usando a técnica das variáveis aleatórias indicadoras (veja a Seção 5.2), seja $X_i = I$\{a i-ésima tentativa de Bernoulli é um sucesso\} para $i = 1, 2, ...,n$; isto é, X_i é a variável aleatória que é 1 se a i-ésima tentativa de Bernoulli é um sucesso e 0 se ela é um insucesso. Assim,

$$X = \sum_{i=1}^{n} X_i$$

e, por linearidade de esperança,

$$\mu = E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} p_i \ ,$$

,

o que implica

$$X - \mu = \sum_{i=1}^{n} (X_i - p_i)$$

Para avaliar E[$e^{a(X-u)}$], substituímos $X - m$, obtendo

$$\begin{aligned} E[e^{a(X-\mu)}] &= E[e^{a\sum_{i=1}^{n}(X_i - p_i)}] \\ &= E\left[\prod_{i=1}^{n} e^{\alpha(X_i - p_i)}\right] \\ &= \prod_{i=1}^{n} E\left[e^{a(X_i - p_i)}\right], \end{aligned}$$

que decorre de (C.24), já que a independência mútua das variáveis aleatórias X_i implica a independência mútua das variáveis aleatórias $e^{a(X_i - p_i)}$ (veja o Exercício C.3-5). Pela definição de expectativa,

$$\begin{aligned} \mathrm{E}\left[e^{\alpha(X-p_i)}\right] &= e^{\alpha(1-p_i)} p_i + e^{\alpha(0-p_i)} q_i \\ &= p_i e^{\alpha q_i} + q_i e^{-\alpha p^i} \\ &\leq p_i e^{\alpha} + 1 \\ &\leq \exp(p_i e^{\alpha}). \end{aligned} \qquad \text{(C.45)}$$

onde exp(x) denota a função exponencial: exp(x) = e_x. (A desigualdade (C.45) decorre das desigualdades $\alpha > 0$, $q_i \leq 1$, $e^{aq_i} \leq e^a$, e a última linha decorre da desigualdade (3.12).) Consequentemente,

$$\begin{aligned} \mathrm{E}\left[e^{\alpha(X-\mu)}\right] &= \prod_{i=1}^{n} E\left[e^{\alpha(X_i - p_i)}\right] \\ &\leq \prod_{i=1}^{n} \exp(p_i e^{\alpha}) \\ &= \exp\left(\sum_{i=1}^{n} p_i e^{\alpha}\right) \\ &= \exp(\mu e^{\alpha}), \end{aligned} \qquad \text{(C.46)}$$

visto que $\mu = \sum_{i=1}^{n} p_i$. Portanto, pela equação (C.43) e desigualdades (C.44) e (C.46), decorre que

$$Pr\{X - \mu \geq r\} \leq exp(\mu e^{\alpha} - \alpha r) \ . \qquad \text{(C.47)}$$

Escolhendo $a = \ln(r/m)$, (veja o Exercício C.5-7), obtemos

$$
\begin{aligned}
\Pr\{X-\mu \geq r\} &\leq \exp(\mu e^{\ln(r/\mu)} - r\ln(r/\mu)) \\
&= \exp(r - r\ln(r/\mu)) \\
&= \frac{e^r}{(r/\mu)^r} \\
&= \left(\frac{\mu e}{r}\right)^r .
\end{aligned}
$$

Quando aplicado a tentativas de Bernoulli nas quais cada tentativa tem a mesma probabilidade de sucesso, o Teorema C.8 produz o corolário a seguir, que limita a cauda direita de uma distribuição binomial.

Corolário C.9

Considere uma sequência de n tentativas de Bernoulli na qual em cada tentativa ocorra sucesso com probabilidade p e insucesso com probabilidade $q = 1 - p$. Então, para $r > np$,

$$
\begin{aligned}
\Pr\{X - np \geq r\} &= \sum_{k=\lceil np+r \rceil}^{n} b(k;n,p) \\
&= \left(\frac{\mu e}{r}\right)^r .
\end{aligned}
$$

Prova Pela equação (C.37), temos $m = \mathrm{E}[X] = np$.

Exercícios

C.5-1 ★ O que é menos provável: não obter nenhuma cara quando lançamos uma moeda não viciada n vezes ou obter menos de n caras quando lançamos uma moeda $4n$ vezes?

C.5-2 ★ Prove os Corolários C.6 e C.7.

C.5-3 ★ Mostre que

$$
\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k;n,a/(a+1))
$$

para todo $a > 0$ e todo k tal que $0 < k < na/(a+1)$.

C.5-4 ★ Prove que, se $0 < k < np$, onde $0 < p < 1$ e $q = 1 - p$, então

$$
\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np-k}\left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k} .
$$

C.5-5 ★ Mostre que as condições do Teorema C.8 implicam que

$$\Pr\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r .$$

De modo semelhante, mostre que as condições do Corolário C.9 implicam que

$$\Pr\{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r .$$

C.5-6 ★ Considere uma sequência de n tentativas de Bernoulli na qual na i-ésima tentativa, para $i = 1, 2, ..., n$, ocorra sucesso com probabilidade p_i e insucesso com probabilidade $q_i = 1 - p_i$. Seja X a variável aleatória que descreve o número total de sucessos, e seja $m = E[X]$. Mostre que, para $r \geq 0$,

$$\Pr\{X - \mu \geq r\} \leq e^{-r^2/2n}.$$

(*Sugestão*: Prove que $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{\alpha^2/2}$. Depois, siga a estrutura da prova do Teorema C.8 usando essa desigualdade em lugar da desigualdade (C.45).)

C.5-7 ★ Mostre que escolher $a = \ln(r/m)$ minimiza o lado direito da desigualdade (C.47).

Problemas

C-1 ***Bolas e caixas***

Neste problema, investigamos o efeito de várias hipóteses sobre o número de modos de colocar n bolas em b caixas distintas.

a. Suponha que as n bolas sejam distintas e que sua ordem dentro de uma caixa não tenha importância. Demonstre que o número de modos de colocar as bolas nas caixas é b^n.

b. Suponha que as bolas sejam distintas e que as bolas em cada caixa estejam ordenadas. Demonstre que há exatamente $(b + n - 1)!/(b - 1)!$ modos de colocar as bolas nas caixas. (*Sugestão*: Considere o número de modos de arranjar em linha n bolas distintas e $b - 1$ bastões indistintos.)

c. Suponha que as bolas sejam idênticas e, consequentemente, que sua ordem dentro de uma caixa não tenha importância. Mostre que o número de modos de colocar as bolas nas caixas é $\binom{b+n-1}{n}$. (*Sugestão*: Dos arranjos dados na parte (b), quantos se repetem se as bolas forem idênticas?)

d. Suponha que as bolas sejam idênticas e que nenhuma caixa possa conter mais de uma bola, de modo que $n \leq b$. Demonstre que o número de modos de colocar as bolas é $\binom{b}{n}$.

e. Suponha que as bolas sejam idênticas e que nenhuma caixa possa ficar vazia. Considerando que $n \geq b$, mostre que o número de modos de colocar as bolas é $\binom{n-1}{b-1}$.

NOTAS DO APÊNDICE

Os primeiros métodos gerais para resolver problemas de probabilidade foram discutidos em uma famosa correspondência entre B. Pascal e P. de Fermat, que começou em 1654, e em um livro de C. Huygens em 1657. A teoria da probabilidade rigorosa começou com o trabalho de J. Bernoulli em 1713 e de A. De Moivre em 1730. Desenvolvimentos adicionais da teoria foram propostos por P.-S. Laplace, S.-D. Poisson e C. F. Gauss.

Somas de variáveis aleatórias foram estudadas originalmente por P. L. Chebyshev e A. A. Markov. Os axiomas da teoria da probabilidade foram desenvolvidos por A. N. Kolmogorov em 1933. Chernoff [66] e Hoeffding [173] determinaram os limites para caudas de distribuições. Um trabalho seminal sobre estruturas combinatórias aleatórias foi realizado por P. Erdös.

Knuth [209] e Liu [237] são boas referências para análise combinatória elementar e contagem. Livros didáticos padrões como os de Billingsley [46], Chung [67], Drake [95], Feller [104] e Rozanov [300] oferecem introduções abrangentes à probabilidade.

---

[1] Quando se trata de uma distribuição de probabilidade geral, podem existir alguns subconjuntos do espaço amostral S que não são considerados eventos. Essa situação normalmente surge quando o espaço amostral é infinito não contável. O principal requisito para que subconjuntos sejam eventos é que o conjunto de eventos de um espaço amostral seja fechado às operações de tomar o complemento de um evento, formar a união de um número de eventos finito ou contável e tomar a interseção de um número de eventos finito ou contável. A maioria das distribuições de probabilidades que veremos refere-se a espaços amostrais finitos ou contáveis e, de modo geral, consideraremos todos os subconjuntos de um espaço amostral eventos. Uma exceção notável é a distribuição de probabilidade uniforme contínua, que veremos em breve.

D MATRIZES

Matrizes aparecem em numerosas aplicações, incluindo, a ciência da computação, mas não se limitando de modo algum a ela. Se você já estudou matrizes, estará familiarizado com grande parte do material apresentado neste apêndice, mas outras partes podem ser novas. A Seção D.1 abrange definições e operações básicas com matrizes, e a Seção D.2 apresenta algumas propriedades básicas de matrizes.

D.1 MATRIZES E OPERAÇÕES COM MATRIZES

Nesta seção, revisamos alguns conceitos básicos da teoria de matrizes e algumas propriedades fundamentais de matrizes.

Matrizes e vetores

Uma matriz é um arranjo retangular de números. Por exemplo,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix} \qquad \text{(D.1)}$$

é uma matriz 2 × 3 $A = (a_{ij})$ onde, para $i = 1, 2$ e $j = 1, 2, 3$, denotamos o elemento da matriz na linha i e coluna j por a_{ij}. Usamos letras maiúsculas para denotar matrizes e índices em letras minúsculas correspondentes às linhas e colunas para denotar seus elementos. Denotamos o conjunto de todas as matrizes $m \times n$ com entradas de valores reais por $^{m \times}{}_{n}$ e, em geral, o conjunto de matrizes $m \times n$ com entradas retiradas de um conjunto S por $S_{m \times n}$.

A ***transposta*** de uma matriz A é a matriz A_T obtida pela troca das linhas e colunas de A. Para a matriz A da equação (D.1),

$$A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Um ***vetor*** é um arranjo unidimensional de números. Por exemplo,

é um vetor de tamanho 3. Às vezes, denominamos um vetor de comprimento ***n-vetor***. Usamos letras minúsculas para denotar vetores e denotamos o i-ésimo elemento de um vetor x de tamanho n por x_i, para $i = 1, 2, ..., n$. Tomamos a forma-padrão de um vetor como um ***vetor coluna*** equivalente a uma matriz $n \times 1$; o ***vetor linha*** correspondente é obtido tomando a transposta:

$$x^{\mathrm{T}} = (\,2 \;\; 3 \;\; 5\,)\,.$$

O ***vetor unitário*** e_i é o vetor cujo i-ésimo elemento é 1 e todos os outros elementos são iguais a zero. Em geral, o tamanho de um vetor unitário fica claro no contexto.

Uma ***matriz nula*** é uma matriz na qual todas as entradas são zero. Tal matriz é frequentemente denotada por 0, já que a ambiguidade entre o número zero e uma matriz de zeros em geral é resolvida facilmente pelo contexto. Se pretendemos uma matriz de zeros, então o tamanho da matriz também precisa ser deduzido do contexto.

Matrizes quadradas

Matrizes quadradas $n \times n$ surgem frequentemente. Diversos casos especiais de matrizes quadradas têm interesse particular:

1. Uma ***matriz diagonal*** tem $a_{ij} = 0$ sempre que $i \neq j$. Como todos os elementos fora da diagonal são zero, a matriz pode ser especificada por uma lista de elementos ao longo da diagonal:

$$\operatorname{diag}(a_{11}, a_{11} \cdots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}.$$

2. A ***matriz identidade*** $n \times n$ I_n é uma matriz diagonal com 1s ao longo da diagonal:

$$I_n = \operatorname{diag}(1,\ldots,1)$$

$$= \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Quando I aparece sem índice, deduzimos seu tamanho do contexto. A i-ésima coluna de uma matriz identidade é o vetor unitário e_i.

3. Uma ***matriz tridiagonal*** T é tal que $t_{ij} = 0$ se $|i - j| > 1$. Entradas não nulas aparecem somente na diagonal principal, imediatamente acima da diagonal principal ($t_{i,\,i+1}$ para $i = 1, 2, ..., n - 1$) ou imediatamente abaixo da diagonal principal ($t_{i+1,\,i}$ para $i = 1, 2, ..., n - 1$):

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \cdots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \cdots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \cdots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-,1,n} \\ 0 & 0 & 0 & 0 & \cdots & 0 & t_{n,n-1} & t_{n-2,n-1} \end{pmatrix}.$$

4. Uma ***matriz triangular superior*** U é tal que $u_{ij} = 0$ se $i > j$. Todas as entradas abaixo da diagonal são zero:

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}.$$

Uma matriz triangular superior é ***triangular superior unitária*** se tem apenas 1s ao longo da diagonal.

5. Uma ***matriz triangular inferior*** L é tal que $l_{ij} = 0$ se $i < j$. Todas as entradas acima da diagonal são zero:

$$L = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix}.$$

Uma matriz triangular inferior é ***triangular inferior unitária*** se tem somente 1s ao longo da diagonal.

6. Uma ***matriz de permutação*** P tem exatamente um 1 em cada linha ou coluna e zeros em todas as outras posições. Um exemplo de matriz de permutação é

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Tal matriz é denominada matriz de permutação porque multiplicar um vetor x por uma matriz de permutação tem o efeito de permutar (rearranjar) os elementos de x. O Exercício D.1-4 explora propriedades adicionais da matrizes de permutação.

7. Uma ***matriz simétrica*** A satisfaz a condição $A = A^T$. Por exemplo,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

é uma matriz simétrica.

Operações básicas com matrizes

Os elementos de uma matriz ou vetor são números que pertencem a um sistema numérico, como os números reais, os números complexos ou inteiros módulo um número primo. O sistema numérico define como somar e multiplicar números. Podemos estender essas definições para englobar a adição e a multiplicação de matrizes.

Definimos a ***adição de matrizes*** da seguinte maneira: se $A = (a_{ij})$ e $B = (b_{ij})$ são matrizes $m \times n$, então a soma dessas matrizes $C = (c_{ij}) = A + B$ é a matriz $m \times n$ definida por

$$c_{ij} = a_{ij} + b_{ij}$$

para $i = 1, 2, ..., m$ e $j = 1, 2, ..., n$. Isto é, a adição de matrizes é executada componente a componente. Uma matriz zero é a identidade para adição de matrizes:

$$A + 0 = A = 0 + A$$

Se l é um número e $A = (a_{ij})$ é uma matriz, então $lA = (la_{ij})$ é o ***múltiplo escalar*** de A obtido pela multiplicação de cada um de seus elementos por l. Como caso especial, definimos a ***negativa*** de uma matriz $A = (a_{ij})$ como $-1 \cdot A = -A$, de modo que a ij-ésima entrada de $-A$ é $-a_{ij}$. Assim,

$$A + (-A) = 0 = (-A) + A.$$

Usamos a negativa de uma matriz para definir ***subtração de matrizes*** $A - B = A + (-B)$.

Definimos ***multiplicação de matrizes*** da maneira descrita a seguir. Começamos com duas matrizes A e B que são ***compatíveis*** no sentido de que o número de colunas de A é igual ao número de linhas de B. (Em geral, sempre consideramos que uma expressão que contém um produto de matrizes AB implica que as matrizes A e B são compatíveis.) Se $A = (a_{ik})$ é uma matriz $m \times n$ e $B = (b_{kj})$ é uma matriz $n \times p$, então seu produto de matrizes $C = AB$ é a matriz $m \times p$ $C = (c_{ij})$, onde

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad \text{(D.2)}$$

para $i = 1, 2, ..., m$ e $j = 1, 2, ..., p$. O procedimento SQUARE-MATRIX-MULTIPLY na Seção 4.2 implementa a multiplicação de matrizes da maneira direta baseada na equação (D.2), considerando que as matrizes sejam quadradas: $m = n = p$. Para multiplicar matrizes $n \times n$, SQUARE-MATRIX-MULTIPLY executa n_3 multiplicações e $n_2(n - 1)$ adições, e seu tempo de execução é $Q(n_3)$.

Matrizes têm muitas (mas não todas) das propriedades algébricas típicas de números. Matrizes identidade são identidades para multiplicação de matrizes:

$$I_m A = AI_n = A$$

para qualquer matriz A $m \times n$. Multiplicar por uma matriz zero dá uma matriz zero:

$$A0 = 0 .$$

A multiplicação de matrizes é associativa:

$$A(BC) = (AB)C$$

para matrizes compatíveis A, B e C. A multiplicação de matrizes é distributiva para a adição:

$$A(B + C) = AB + AC ,$$
$$(B + C)D = BD + CD .$$

Para $n > 1$, a multiplicação de matrizes $n \times n$ não é comutativa. Por exemplo, se

$$A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \text{ e } B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \text{então}$$

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

e

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

Definimos produtos matriz-vetor ou produtos vetor-vetor como se o vetor fosse a matriz $n \times 1$ equivalente (ou uma matriz $1 \times n$, no caso de um vetor linha). Assim, se A é uma matriz $m \times n$ e x é um n-vetor, então Ax é um m-vetor. Se x e y são vetores n, então

$$x^T y = \sum_{i=1}^{n} x_i y_i$$

é um número (na realidade, uma matriz 1×1) denominado ***produto interno*** de x e y. A matriz xy_T é uma matriz $n \times n$ Z denominada ***produto externo*** de x e y, com $z = x\ y$. A ***norma (euclidiana)*** $\|x\|$ de um vetor n x é definida por

$$\begin{aligned}\|x\| &= (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2} \\ &= (x^T x)^{1/2}\end{aligned}$$

Assim, a norma de x é seu comprimento no espaço euclidiano de n dimensões.

Exercícios

D.1-1 Mostre que, se A e B são matrizes $n \times n$ simétricas, então $A + B$ e $A - B$ também são simétricas.

D.1-2 Prove que $(AB)^T = B_T A_T$ e que $A_T A$ é sempre uma matriz simétrica.

D.1-3 Prove que o produto de duas matrizes triangulares inferiores é triangular inferior.

D.1-4 Prove que, se P é uma matriz de permutação $n \times n$ e A é uma matriz $n \times n$, então o produto de matrizes PA é A com suas linhas permutadas e que o produto de matrizes AP é A com suas colunas permutadas. Prove que o produto de duas matrizes de permutação é uma matriz de permutação.

D.2 PROPRIEDADES BÁSICAS DE MATRIZES

Nesta seção, definimos algumas propriedades básicas pertinentes a matrizes: inversas, dependência linear, independência linear, posto e determinantes. Definimos também a classe de matrizes positivas definidas.

Inversas, postos e determinantes de matrizes

Definimos a ***inversa*** de uma matriz A $n \times n$ como a matriz $n \times n$ denotada por A^{-1} (se existir), tal que $AA^{-1} = I_n = A^{-1}A$. Por exemplo,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}.$$

Muitas matrizes $n \times n$ não nulas não têm inversas. Uma matriz que não tem inversa é denominada ***não inversível*** ou ***singular***. Um exemplo de matriz singular não nula é

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

Se uma matriz tem uma inversa, ela é denominada ***inversível*** ou ***não singular***. Inversas de matrizes, quando existem, são únicas. (Veja o Exercício D. 2-1.) Se A e B são matrizes $n \times n$ não singulares, então

$$(BA)^{-1} = A^{-1} B^{-1} .$$

A operação inversa comuta com a operação transposta:

$$(A^{-1})^T = (A^T)^{-1} .$$

Os vetores $x_1, x_2, ..., x_n$ são ***linearmente dependentes*** se existem coeficientes $c_1, c_2, ..., c_n$, nem todos iguais a zero, tais que $c_1x_1 + c_2x_2 + ... + c_nx_n = 0$. Por exemplo, os vetores linhas $x_1 = (\ 1\ 2\ 3\ )$, $x_2 = (\ 2\ 6\ 4\ )$ e $x_3 = (\ 4\ 11\ 9\ )$

são linearmente dependentes, já que $2x_1 + 3x_2 - 2x_3 = 0$. Se os vetores não são linearmente dependentes, são ***linearmente independentes***. Por exemplo, as colunas de uma matriz identidade são linearmente independentes.

O ***posto coluna*** de uma matriz A $m \times n$ não nula é o tamanho do maior conjunto de colunas linearmente independentes de A. De modo semelhante, o ***posto linha*** de A é o tamanho do maior conjunto de linhas linearmente independentes de A. Uma propriedade fundamental de qualquer matriz A é que seu posto linha é sempre igual a seu posto coluna, de modo que podemos simplesmente nos referir ao ***posto*** de A. O posto de uma matriz $m \times n$ é um inteiro entre 0 e $\min(m, n)$, inclusive. (O posto de uma matriz zero é 0, e o posto de uma matriz identidade $n \times n$ é n.) Uma definição alternativa, embora equivalente e, muitas vezes, mais útil é que o posto de uma matriz A $m \times n$ não nula é o menor número r tal que existem matrizes B e C de tamanhos respectivos $m \times r$ e $r \times n$ tais que

$$A = BC .$$

Uma matriz quadrada $n \times n$ tem ***posto completo*** se seu posto é n. Uma matriz $m \times n$ tem ***posto coluna completo*** se seu posto é n. O teorema a seguir dá uma propriedade fundamental de postos.

Teorema D.1

Uma matriz quadrada tem posto completo se e somente se ela é não singular.

Um ***vetor anulador*** para uma matriz A é um vetor não nulo x tal que $Ax = 0$. O teorema a seguir, cuja prova fica para o Exercício D.2-7 e seu corolário relacionam as noções de posto coluna e singularidade a vetores anuladores.

Teorema D.2

Uma matriz A tem posto coluna completo se e somente se não tem vetor anulador.

Corolário D.3

Uma matriz quadrada A é singular se e somente se tem vetor anulador.

O ij-ésimo ***menor*** de uma matriz $n \times n$ A, para $n > 1$ é a matriz $(n - 1) \times (n - 1)$ A_{ij} obtida pela eliminação da i-ésima linha e da j-ésima coluna de A. Definimos o ***determinante*** de uma matriz $n \times n$ A recursivamente em termos de seus menores por

$$\det(A) = \begin{cases} a_{11} & \text{se } n = 1 \\ \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(A[1j]) & \text{se } n > 1 \end{cases}$$

O termo $(-1)^{i+j} \det(A\ )$ é conhecido como ***cofator*** do elemento a_{ij}.

Os teoremas a seguir, cujas provas são omitidas aqui, expressam propriedades fundamentais do determinante.

Teorema D.4 (Propriedades de determinantes)

O determinante de uma matriz quadrada A tem as seguintes propriedades:

- Se qualquer linha ou qualquer coluna de A é nula, então $\det(A) = 0$.
- O determinante de A é multiplicado por l se as entradas de qualquer uma das linhas (ou de qualquer uma das colunas) de A são multiplicadas por l.
-

- O determinante de A não se altera se as entradas em uma linha (respectivamente, coluna) são adicionadas às de outra linha (respectivamente, coluna).
- O determinante de A é igual ao determinante de A^T.
- O determinante de A é multiplicado por -1 se quaisquer duas linhas (respectivamente, colunas) são trocadas.

Além disso, para quaisquer matrizes quadradas A e B, temos $\det(AB) = \det(A)\det(B)$.

Teorema D.5

Uma matriz $n \times n$ A é singular se e somente se $\det(A) = 0$.

Matrizes positivas definidas

Matrizes positivas definidas desempenham um papel importante em muitas aplicações. Uma matriz A $n \times n$ é ***positiva definida*** se $x^T Ax > 0$ para todos os n-vetores $x \neq 0$. Por exemplo, a matriz identidade é positiva definida, já que, para qualquer vetor não nulo $x = (x_1\, x_2 \ldots x_n)^T$,

$$\begin{aligned} x^T I_n x &= x^T x \\ &= \sum_{i=1}^{n} x_i^2 \\ &> 0. \end{aligned}$$

Matrizes que surgem em aplicações frequentemente são positivas definidas devido ao teorema a seguir.

Teorema D.6

Para qualquer matriz A com posto coluna completo, a matriz $A^T A$ é positiva definida.

Prova Devemos mostrar que $x^T(A^T A)x > 0$ para qualquer vetor não nulo x. Para qualquer vetor x,

$$\begin{aligned} x^T(A^T A)x &= (Ax)^T(Ax) \quad \text{(pelo Exercício D.1-2)} \\ &= ||Ax||^2 \end{aligned}$$

Observe que $||Ax||^2$ é apenas a soma dos quadrados dos elementos do vetor Ax. Então, $||Ax||^2. \geq 0$. Se $||Ax||^2 = 0$, todo elemento de Ax é 0, o que significa que $Ax = 0$. Visto que A tem posto coluna completa, $Ax = 0$ implica $x = 0$, pelo Teorema D.2. Consequentemente, $A^T A$ é positiva definida.

A Seção 28.3 explora outras propriedades de matrizes positivas definidas.

Exercícios

D.2-1 Prove que matrizes inversas são únicas, isto é, se B e C são inversas de A, então $B = C$.

D.2-2 Prove que o determinante de uma matriz triangular inferior ou triangular superior é igual ao produto dos elementos de sua diagonal. Prove que a inversa de uma matriz triangular inferior, se existir, é triangular inferior.

D.2-3 Prove que, se P é uma matriz de permutação, então P é inversível, sua inversa é PT, e PT é uma matriz permutação.

D.2-4 Sejam A e B matrizes $n \times n$ tais que $AB = I$. Prove que, se A' é obtida de A mediante a adição da linha j à linha i, então subtrair a coluna i da coluna j de B produz a inversa B' de A'.

D.2-5 Seja A uma matriz não singular $n \times n$ com entradas complexas. Mostre que toda entrada de A^{-1} é real se e somente se toda entrada de A é real.

D.2-6 Mostre que, se A é uma matriz não singular simétrica $n \times n$, então A^{-1} é simétrica. Mostre que, se B é uma matriz arbitrária $m \times n$, então a matriz $m \times m$ dada pelo produto BAB_T é simétrica.

D.2-7 Prove o Teorema D.2. Isto é, mostre que uma matriz A tem posto coluna completo se e somente se $Ax = 0$ implica $x = 0$. (*Sugestão*: Expresse a dependência linear de uma coluna em relação às outras como uma equação matriz-vetor.)

D.2-8 Prove que, para quaisquer duas matrizes compatíveis A e B,

$$\text{posto}(AB) \leq \min(\text{posto}(A), \text{posto}(B)) ,$$

onde a igualdade se mantém se A ou B é uma matriz quadrada não singular. (*Sugestão*: Use a definição alternativa de posto de uma matriz.)

Problemas

D-1 Matriz de Vandermonde

Dados os números $x_0, x_1, ..., x_{n-1}$, prove que o determinante da ***matriz de Vandermonde***

$$V\left(x_0, x_1, \cdots x_{n-1}\right)\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_{0-1}^n \\ 1 & x_1 & x_1^2 & \cdots & x_{0-1}^n \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

é

$$\det(V(x_0, x_1, ..., x_{n-1})) = \prod_{0<i<k<n-1} (x_k - x_j) .$$

(*Sugestão*: Multiplique a coluna i por $-x_0$ e adicione-a à coluna $i + 1$ para $i = n - 1, n, n - 2, ..., 1$, e depois use indução.)

D-2 Permutações definidas por multiplicação matriz-vetor em GF(2)

Uma classe de permutações de inteiros no conjunto $S_n = \{0, 1, 2,..., 2^n - 1\}$ é definida por multiplicação de matrizes em $GF(2)$. Para cada inteiro x em S_n, vemos sua representação binária como um vetor de n bits

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

onde $x = \sum_{i=0}^{n-1} x_i 2^i$. Se A é uma matriz $n \times n$ na qual cada entrada é zero ou 1, então podemos definir uma permutação mapeando cada valor x em S_n para o número cuja representação binária é o produto matriz-vetor Ax. Aqui, efetuamos toda a aritmética em ***GF*(*2*)**: todos os valores são 0 ou 1 e, com uma única exceção, as regras usuais da adição e da multiplicação se aplicam. A exceção é que 1 + 1 = 0. Você pode muito bem imaginar que a aritmética em ***GF*(*2*)** é exatamente igual à aritmética normal de inteiros, exceto que usará somente o bit menos significativo.

Como exemplo, para $S_2 = \{0, 1, 2, 3\}$, a matriz

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

define a seguinte permutação A: $A(0) = 0$, $A(1) = 3$, $A(2) = 2$, $A(3) = 1$. Para ver por que $A(3) = 1$, observe que, trabalhando em $GF.2/$,

$$\begin{aligned} \pi_A(3) &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \cdot 1 & + & 0 \cdot 1 \\ 1 \cdot 1 & + & 1 \cdot 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{aligned}$$

que é a representação binária de 1.

No restante deste problema, trabalhamos com $GF(2)$ e todas as entradas de matrizes e vetores são 0 ou 1. Definimos a o posto de uma matriz 0-1 (uma matriz para a qual cada entrada é 0 ou 1) em $GF(2)$ do mesmo modo que para uma matriz comum, porém com toda a aritmética que determina dependência linear executada em $GF(2)$. Definimos a ***imagem*** de uma matriz 0-1 $n{\times}n$ A por

de modo que $R(A)$ seja o conjunto de números em S_n que podemos produzir multiplicando cada valor de x em S_n por por A.

a. Se r é o posto da matriz A, prove que $|R(A)| = 2^r$. Conclua que A define uma permutação em S_n somente se A tiver posto completo.

Para uma matriz $n \times n$ A dada e para um valor $y \in R(A)$ dado, definimos a ***pré-imagem de*** y por

$$P(A, y) = \{x : Ax = y\} \ ;$$

de modo que $P(A, y)$ seja um conjunto de valores em S_n que mapeia para y quando multiplicado por A.

b. Se r é o posto da matriz $n \times n$ A e $y \in R(A)$, prove que $|P(A, y)| = 2^{n-r}$.

Seja $0 \leq m \leq n$, e suponha que particionamos o conjunto S_n em blocos de números consecutivos, onde o i-ésimo bloco consiste nos 2^m números $i2^m$, $i2^m + 1$, $i2^m + 2, \ldots, (i + 1)2^m - 1$. Para qualquer subconjunto $S \subseteq S_n$, defina $B(S,m)$ como o conjunto de blocos de tamanho 2^m de S_n que contém algum elemento de S. Como exemplo, quando $n = 3$, $m = 1$ e $S = \{1, 4, 5\}$, então $B(S,m)$ consiste em blocos 0 (visto que 1 está no 0-ésimo bloco) e 2 (visto que 4 e 5 estão no bloco 2).

c. Seja r o posto da submatriz inferior esquerda $(n - m) \times m$ de A, isto é, a matriz formada tomando a interseção das $n - m$ linhas inferiores com as m colunas da extrema esquerda de A. Seja S qualquer bloco de tamanho 2^m de S_n e seja $S' = \{y : y = Ax$ para algum $x \in S\}$. Prove que $|B(S', m)| = 2^r$ e que, para cada bloco em $B(S',m)$, exatamente 2^{m-r} números em S mapeiam para aquele bloco.

Como multiplicar o vetor zero por qualquer matriz produz um vetor zero, o conjunto de permutações de S_n definido pela multiplicação de matrizes 0-1 $n \times n$ com posto completo em $GF(2)$ não pode incluir todas as permutações de S_n. Vamos estender a classe de permutações definida pela multiplicação matriz-vetor para incluir um termo aditivo, de modo que $x \in S_n$ mapeie para $Ax + c$, onde c é um vetor de n bits e a adição é efetuada em $GF(2)$. Por exemplo, quando

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

e

$$c = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

obtemos a seguinte permutação ${}_{A,c} : {}_{A,c}(0) = 2, {}_{A,c}(1) = 1, {}_{A,c}(2) = 0, {}_{A,c}(3) = 3$. Denominamos ***permutação linear*** qualquer permutação que mapeie $x \in S_n$ para $Ax + c$, para alguma matriz 0-1 $n \times n$ A com posto completo e algum vetor c de n bits .

d. Use um argumento de contagem para mostrar que o número de permutações lineares de S_n é muito menor que o número de permutações de S_n.

e. Dê um exemplo de valor de n e uma permutação de S_n que não possa ser conseguida por nenhuma permutação linear. (*Sugestão*: Para dada permutação, pense em como multiplicar uma matriz por um vetor unitário está relacionado com as colunas da matriz.)

Notas do Apêndice

Livros didáticos de álgebra linear apresentam muitas informações sobre matrizes. Os livros de Strang [323, 324] são particularmente bons.

BIBLIOGRAFIA

[1] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions.* Dover, 1965.

[2] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady,* 3(5): 1259–1263, 1962.

[3] Leonard M. Adleman, Carl Pomerance, and Robert S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics,* 117: 173–206, 1983.

[4] Alok Aggarwal and Jeffrey Scott Vitter. The inputJoutput complexity of sorting and related prob1ems. *Communications ef the ACM,* 31(9):1116–1127, 1988.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[7] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[8] Ravindra K. Ahuja, Kurt Mehlhom, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM,* 37:213– 223, 1990.

[9] Ravindra K. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research,* 37(5):748–759, 1989.

[10] Ravindra K. Ahuja, James B. Orlin, and Robert E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing,* 18(5):939–954, 1989.

[11] Miklós Ajtai, Nimrod Megiddo, and Orli Waarts. Improved algorithms and analysis for secretary problems and generalizations. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science,* pages 473–482, 1995.

[12] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[13] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications,* 10(2):195–210, 1998.

[14] Noga Alon. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters,* 35:201–204, 1990.

[15] Arne Andersson. Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures,* volume 709 of *Lecture Notes in Computer Science,* pages 60–71. Springer, 1993.

[16] Arne Andersson. Faster detenninistic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science,* pages 135–141, 1996.

[17] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences,* 57:74–93, 1998.

[18] Tom M. Apostol. *Calculus,* volume 1. Blaisdell Publishing Company, second edition, 1967.

[19] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures,* pages 119–129, 1998.

[20] Sanjeev Arora. *Probabilistic checking of proofs and the hardness of approximation problems.* PhD thesis, University of Califomia, Berkeley, 1994.

[21] Sanjeev Arora. The approximability of NP-hard problems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing,* pages 337–348, 1998.

[22] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM,* 45(5):753–782, 1998.

[23] Sanjeev Arora and Carsten Lund. Hardness of approximations. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems,* pages 399–446. PWS Publishing Company, 1997.

[24] Javed A. Aslam. A simple bound on the expected height of a randomly built binary search tree. Technical Report TR2001-387, Dartmouth College Department of Computer Science, 2001.

[25] Mikhail J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[26] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Proble- ms and Their Approximability Properties.* Springer, 1999.

[27] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. *ACM Transactions on Graphics,* 26(3), article 10, 2007.

[28] Sara Baase and Alan Van Gelder. *Computer Algorithms: lntroduction to Design and Analysis.* Addison-Wesley, third edition, 2000.

[29] Eric Bach. Private communication, 1989.

[30] Eric Bach. Number-theoretic algorithms. In *Annual Review of Computer Science,* volume 4, pages 119– 172. Annual Reviews, Inc., 1990.

[31] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory – Volume 1: Eificient Algorithms.* The MIT Press, 1996.

[32] David H. Bailey, King Lee, and Horst D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing,* 4(4):357–371, 1990.

[33] Surender Baswana, Ramesh Hariharan, and Sandeep Seno Improved decremental algorithms for maintaining transitive dosure and all-pairs shortest paths. *Journal of Algorithms,* 62(2):74–92, 2007.

[34] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica,* 1(4):290–306, 1972.

[35] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta lnformatica,* 1(3):173–189, 1972.

[36] Pierre Beauchemin, Gilles Brassard, Claude Crépeau, Claude Goutier, and Carl Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology,* 1 (l):5364, 1988.

[37] Richard Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[38] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics,* 16(1):87–90, 1958.

[39] Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing,* pages 80–86, 1983.

[40] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science,* pages 399–409, 2000.

[41] Samuel W. Bent and John W. John. Finding the median requires $2n$ comparisons. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing,* pages 213–216, 1985.

[42] Jon L. Bentley. *Writing Eificient Programs.* Prentice Hall, 1982.

[43] Jon L. Bentley. *Programming Pearls.* Addison-Wesley, 1986.

[44] Jon L. Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divideand-conquer recurrences. *SIGACT News,* 12(3):36–44, 1980.

[45] Daniel Bienstock and Benjamin McClosky. Tightening simplex mixed-integer sets with guaranteed bounds. *Optimization Online,* July 2008.

[46] Patrick Billingsley. *Probability and Measure.* John Wiley & Sons, second edition, 1986.

[47] Guy E. Blelloch. *Scan Primitives and Parallel Vector Models.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1989. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-463.

[48] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM,* 39(3):85–97, 1996.

[49] Guy E. BlelIoch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures,* pages 1–12, 1995.

[50] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences,* 7(4):448–461, 1973.

[51] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. RandalI, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing,* 37(1):55–69, 1996.

[52] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Joumal on Computing,* 27(1):202–229, 1998.

[53] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Joumal of the ACM,* 46(5):720–748, 1999.

[54] Béla Bollobás. *Random Graphs.* Academic Press, 1985.

[55] Gilles Brassard and Paul Bratley. *Fundamentais of Algorithmics.* Prentice Hall, 1996.

[56] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Joumal of the ACM,* 21(2):201–206, 1974.

[57] Richard P. Brent. An improved Monte Carlo factorization algorithm. BIT, 20(2):176–184, 1980.

[58] J. P. Buhler, H. W. Lenstra, Jr., and Carl Pomerance. Factoring integers with the number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve,* volume 1554 of *Lecture Notes in Mathematics,* pages 50–94. Springer, 1993.

[59] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Joumal of Computer and System Sciences,* 18(2):143–154, 1979.

[60] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press, 2007.

[61] Bemard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Joumal of the ACM,* 47(6):1028–1047, 2000.

[62] Joseph Cheriyan and Torben Hagerup. A randomized maximum-fiow algorithm. *SIAM Joumal on Computing,* 24(2):203–226, 1995.

[63] Joseph Cheriyan and S. N. Maheshwari. Ana1ysis of prefiow push algorithms for maximum network fiow. *SIAM Joumal on Computing,* 18(6):1057–1086, 1989.

[64] Boris V. Cherkassky and Andrew V. Goldberg. On implementing the push-relabel method for the maximum fiow problem. *Algorithmica,* 19(4):390–410, 1997.

[65] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths aIgorithms: Theory and experimental evaluation. *Mathematical Programming,* 73(2):129–174, 1996.

[66] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists and monotone priority queues. *SIAM Joumal on Computing,* 28(4):1326–1346, 1999.

[67] H. Chemoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics,* 23(4):493– 507, 1952.

[68] Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes.* Springer, 1974.

[69] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research,* 4(3):233–235, 1979.

[70] V. Chvátal. *Linear Programming.* W. H. Freeman and Company, 1983.

[71] V. Chvátal, D. A. Klamer, and D. E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, 1972.

[72] Cilk Arts, Inc., Burlington, Massachusetts. *Cilk++ Programmer's Guide,* 2008. Available at http://www.cilk.comlarchive/docs/cilkIguide.

[73] Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 Congressfor Logic, Methodology, and the Philosophy of Science,* pages 24–30. North-Holland, 1964.

[74] H. Cohen and H. W. Lenstra, Jr. Primality testing and Jacobi sums. *Mathematics of Computation,* 42(165):297–330, 1984.

[75] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys,* 11(2):121–137, 1979.

[76] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing,* pages 151–158, 1971.

[77] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation,* 19(90):297–301, 1965.

[78] Don Coppersmith. Modifications to the number field sieve. *Journal of Cryptology,* 6(3):169–180, 1993.

[79] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation,* 9(3):251–280, 1990.

[80] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing,* 28(1):105–136, 1998.

[81] Don Dailey and Charles E. Leiserson. Using Cilk to write multiprocessor chess programs. In H. J. van den Herik and B. Monien, editors, *Advances in Computer Games,* volume 9, pages 25–52. University of Maastricht, Netherlands, 200l.

[82] Paolo D'Alberto and Alexandru Nicolau. Adaptive Strassen's matrix multiplication. In *Proceedings of the 21st Annuallnternational Conference on Supercomputing,* pages 284–292, June 2007.

[83] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms.* McGraw-Hill, 2008.

[84] Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. Engineering a sorted list data structure for 32 bit keys. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics,* pages 142–151, January 2004.

[85] Camil Demetrescu and Giuseppe F. Italiano. Fully dynamic ali pairs shortest paths with real edge weights. *Journal of Computer and System Sciences,* 72(5):813–837, 2006.

[86] Eric V. Denardo and Bennett L. Fox. Shortest-route methods: l. Reaching, pruning, and buckets. *Operations Research,* 27(1):161–186, 1979.

[87] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhom, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journalon Computing,* 23(4):738–761, 1994.

[88] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory,* IT-22(6):644–654, 1976.

[89] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik,* 1(1):269– 271, 1959.

[90] E. A. Dinic. Algorithm for solution of a problem of maximum ftow in a network with power estimation. *Soviet Mathematics Doklady,* 11(5):1277–1280, 1970.

[91] Brandon Dixon, Monika Rauch, and Robert E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing,* 21(6): 1184–1192, 1992.

[92] John D. Dixon. Factorization and primality tests. *The American Mathematical Monthly,* 91(6):333– 352, 1984.

[93] Dorit Dor, Johan Hastad, Staffan Ulfberg, and Uri Zwick. On lower bounds for selecting the mediano *SIAM Journal on Discrete Mathematics,* 14(3):299– 311,2001.

[94] Dorit Dor and Uri Zwick. Selecting the mediano *SIAM Journal on Computing,* 28(5): J722–1758, 1999.
[95] Dorit Dor and Uri Zwick. Median selection requires (2 +)n comparisons. *SIAM Journal on Discrete Mathematics,* 14(3):312–325,2001.
[96] A1vin W. Drake. *Fundamentais of Applied Probability Theory.* McGraw-Hill, 1967.
[97] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An altemative to Fibonacci heaps with app1ications to paralle1 computation. *Communications of the ACM,* 31(11):1343–1354, 1988.
[98] James R. Driscoll, Neil Samak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences,* 38(1): 86–124, 1989.
[99] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry,* volume 10 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1987.
[100] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics,* 17:449–467, 1965.
[101] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming,* 1 (1): 127–136, 1971.
[102] Jack Edmonds and Richard M. Karp. Theoretical improvements in the algorithrnic efficiency for network ftow problems. *Journal of the ACM,* 19(2):248–264, 1972.
[103] Shimon Even. *Graph AIgorithms.* Computer Science Press, 1979.
[104] William Feller. *An Introduction to Probability Theory and Its Applications.* John Wiley & Sons, third edition, 1968.
[105] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM,* 5(6):345, 1962.
[106] Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM,* 7(12):701, 1964.
[107] Robert W. F1oyd. Permuting information in idealized two-level storage. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations,* pages 105–109. Plenum Press, 1972.
[108] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM,* 18(3):165–172, 1975.
[109] Lestor R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.
[110] Lestor R. Ford, Jr. and Selmer M. Johnson. A toumament problem. *The American Mathematical Monthly,* 66(5):387–389, 1959.
[111] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing,* 5(1):83–89, 1976.
[112] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM,* 31(3):538– 544, 1984.
[113] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing,* pages 345–354, 1989.
[114] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM,* 34(3):596– 615, 1987.
[115] Michael L. Fredman and Dan E. Willard. Surpassing the infornation theoretic bound with fusion trees. *Journal of Computer and System Sciences,* 47(3):424–436, 1993.
[116] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.
[117] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE,* 93(2):216–231, 2005.
[118] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the* 1998 *ACM SIGPLAN Conference on Programming Language Design and 1mplementation,* pages 212–223, 1998.
[119] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters,* 74(3–4): 107–114, 2000.
[120] Harold N. Gabow, Z. Galil, T. Spencer, and Robert E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica,* 6(2): 109–122, 1986.
[121] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences,* 30(2):209–221, 1985.
[122] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing,* 18(5):1013–1036, 1989.
[123] Zvi Galil and Oded Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation,* 134(2):103–139, 1997.
[124] Zvi Galil and Oded Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences,* 54(2):243– 254, 1997.
[125] Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science,* 92(1):49–76, 1992.
[126] Zvi Galil and Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences,* 26(3):280–294, 1983.
[127] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms,* pages 165–174, 1993.
[128] Michael R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analyis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing,* pages 143–150, 1972.
[129] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[130] Saul Gass. *Linear Programming: Methods and Applications.* International Thomson Publishing, fourth edition, 1975.

[131] Fánicá Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing,* 1(2):180–187, 1972.

[132] Alan George and Joseph W-H Liu. *Computer Solu- tion of Large Sparse Positive Definite Systems.* Prentice Hall, 1981.

[133] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bel! System Technical Journal,* 38(4): 933–967, 1959.

[134] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM,* 42(6):1115–1145, 1995.

[135] Michel X. Goemans and David P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In Dorit S. Hochbaum, editor, *Approximation Algorithmsfor NP-Hard Problems,* pages 144–191. PWS Publishing Company, 1997.

[136] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[137] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing,* 24(3):494–504, 1995.

[138] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM,* 45(5):783–797, 1998.

[139] Andrew V. Goldberg, Éva Tardos, and Robert E. Tarjan. Network flow algorithms. In Bemhard Korte, László Lovász, Hans Jürgen Promel, and Alexander Schrijver, editors, *Paths, Flows, and VLSI- -Layout,* pages 101–164. Springer, 1990.

[140] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM,* 35(4):921–940, 1988.

[141] D. Goldfarb and M. J. Todd. Linear programming. In G. L. Nemhauser, A. H. G. RinnooyKan, and M. J. Todd, editors, *Handbook in Operations Research and Management Science, Vol. I, Optimization,* pages 73–170. Elsevier Science Publishers, 1989.

[142] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences,* 28(2):270–299, 1984.

[143] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[144] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, third edition, 1996.

[145] G. H. Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 1984.

[146] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Addison-Wesley, 1992.

[147] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java.* John Woley & Sons, 1998.

[148] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples.* John Wiley & Sons, 2001.

[149] Ronald L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal,* 45(9):1563–1581, 1966.

[150] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters,* 1(4):132–133, 1972.

[151] Ronald L. Graham and Pavol HelI. On the history of the minimum spanning tree problem. *Annals ofthe History of Computing,* 7(1):43–57, 1985.

[152] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, second edition, 1994.

[153] David Gries. *The Science of Programming.* Springer, 1981.

[154] M. Grotschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer, 1988.

[155] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science,* pages 8–21, 1978.

[156] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biolo- gy.* Cambridge University Press, 1997.

[157] H. Halberstam and R. E. Ingram, editors. *The Mathematical Papers of Sir William Rowan Hamilton,* volume III (Algebra). Cambridge University Press, 1967.

[158] Yijie Han. Improved fast integer sorting in linear space. In *Proceedings of the 12th ACMSIAM Symposium on Discrete Algorithms,* pages 793–796, 2001.

[159] Yijie Han. An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for ali pairs shortest path. *Algorithmica,* 51(4):428–434, 2008.

[160] Frank Harary. *Graph Theory.* Addison-Wesley, 1969.

[161] Gregory C. Harfst and Edward M. Reingold. A potential-based amortized analysis of the union-find data structure. *SIGACT News,* 31(3):86–95, 2000.

[162] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society,* 117:285–306, May 1965.

[163] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the Fast Fourier Transform. *IEEE ASSP Magazine,* 1(4):14–21, 1984.

[164] Monika R. Henzinger and VaJerie King. Fu1ly dynarnic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science,* pages 664–672, 1995.

[165] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM,* 46(4):502–516, 1999.

[166] Monika R. Henzinger, Satish Rao, and Harold N. Gabow. Computing vertex connectivity: New bounds from old techniques. *Journal of Algorithms,* 34(2):222–250, 2000.

[167] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transac- tions on Mathematical Software,* 16(4):352–368, 1990.
[168] W. Daniel HiIlis and Jr. Guy L. SteeJe. Data parallel algorithms. *Communications of the ACM,* 29(12):1170–1183, 1986.
[169] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM,* 4(7):321–322, 1961.
[170] C. A. R. Hoare. Quicksort. *Computer Journal,* 5(1): 10–15, 1962.
[171] Dorit S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Mathematics,* 6(3):243–254, 1983.
[172] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, 1997.
[173] W. Hoeffding. On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics,* 27(3):713–721, 1956.
[174] Micha Hofri. *Probabilistic Analysis of Algorithms.* Springer, 1987.
[175] Micha Hofri. *Analysis of Algorithms.* Oxford University Press, 1995.
[176] John E. Hopcroft and Richard M. Karp. An $n5/2$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing,* 2(4):225–231, 1973.
[177] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, third edition, 2006.
[178] John E. Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM,* 16(6):372–378, 1973.
[179] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing,* 2(4):294–303, 1973.
[180] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-WesJey, 1979.
[181] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer Algorithms.* Computer Science Press, 1998.
[182] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing,* 11(2):362Volume 1373, 1982.
[183] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing,* 13(2):228–251, 1984.
[184] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics,* 21(4):514– 532, 1971.
[185] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proeeedings of the IRE,* 40(9): 1098–1101, 1952.
[186] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Tumbull. Implementation of Strassen's algorithm for matrix multiplication. In *Proeeedings of the* 1996 *ACM/IEEE Conferenee on Supereomputing,* article 32, 1996.
[187] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM,* 22(4):463–468, 1975.
[188] E. J. Isaac and R. C. Singleton. Sorting by address ca1culation. *Journal of the ACM,* 3(3):169–174, 1956.
[189] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Proeessing Letters,* 2(1): 18–21, 1973.
[190] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Scienees,* 9(3):256–278, 1974.
[191] David S. Johnson. The NP-completeness column: An ongoing guide-The tale of the second prover. *Journal of Algorithms,* 13(3):502–524, 1992.
[192] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM,* 24(1):1–13, 1977.
[193] Richard Johnsonbaugh and Marcus Schaefer. *Algorithms.* Pearson Prentice Hall, 2004.
[194] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physies- -Doklady,* 7(7):595–596, 1963. Translation of an article in *Doklady Akademii Nauk SSSR,* 145(2), 1962.
[195] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM,* 42(2):321–328, 1995.
[196] David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM Journal on Computing,* 22(6):1199–1217, 1993.
[197] Howard Karloff. *Linear Programming.* Birkhiiuser, 1991.
[198] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatoriea,* 4(4):373– 395, 1984.
[199] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations,* pages 85–103. Plenum Press, 1972.
[200] Richard M. Karp. An introduction to randomized algorithms. *Diserete Applied Mathematies*, 34(1–3):165–201, 1991.
[201] Richard M. Karp and Michael O. Rabin. Efficient randomized pattem-matching algorithms. *IBM Journal of Researeh and Development,* 31 (2):249– 260, 1987.
[202] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematies Doklady,* 15(2):434–437, 1974.
[203] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmiea,* 18(2):263–270, 1997.

[204] Valerie King, Satish Rao, and Robert E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms,* 17(3):447–474, 1994.
[205] Jeffrey H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis.* Addison-Wesley, second edition, 1997.
[206] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing,* 15(2):287–299, 1986.
[207] Philip N. Klein and Neal E. Young. Approximation algorithms for NP-hard optimization problems. In *CRC Handbook on Algorithms,* pages 34–1–34–19. CRC Press, 1999.
[208] Jon Kleinberg and Éva Tardos. *Algorithm Design.* Addison-Wesley, 2006.
[209] Donald E. Knuth. *Fundamental Algorithms,* volume 1 of *The Art of Computer Programming.* Addison-Wesley, 1968. Third edition, 1997.
[210] Donald E. Knuth. *Seminumerical Algorithms,* volume 2 of *The Art of Computer Programming.* Addison-Wesley, 1969. Third edition, 1997.
[211] Donald E. Knuth. *Sorting and Searching,* volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973. Second edition, 1998.
[212] Donald E. Knuth. Optimum binary search trees. *Acta Iriformatica,* 1(1):14–25, 1971.
[213] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News,* 8(2):18–23, 1976.
[214] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattem matching in strings. *SIAM Journal on Computing,* 6(2):323–350, 1977.
[215] J. Komlós. Linear verification for spanning trees. *Combinatorica,* 5(1):57–65, 1985.
[216] Bernhard Korte and László Lovász. Mathematical structures underlying greedy algorithms. In F. Gecseg, editor, *Fundamentais of Computation Theory,* volume 117 of *Lecture Notes in Computer Science,* pages 205–209. Springer, 1981.
[217] Bemhard Korte and László Lovász. Structural properties of greedoids. *Combinatorica,* 3(3–4):359–374, 1983.
[218] Bemhard Korte and László Lovász. Greedoids–A structural framework for the greedy algorithm. In W. Pulleybank, editor, *Progress in Combinatorial Optimization,* pages 221–243. Academic Press, 1984.
[219] Bemhard Korte and László Lovász. Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods,* 5(2):229–238, 1984.
[220] Dexter C. Kozen. *The Design and Analysis of Algorithms.* Springer, 1992.
[221] David W. Krumme, George Cybenko, and K. N. Venkataraman. Gossiping in mini mal time. *SIAM Journal on Computing,* 21(1):111–139, 1992.
[222] Joseph B. Kruskal, Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society,* 7(1):48–50, 1956.
[223] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers,* C-28(9):690–691, 1979.
[224] Eugene L. LawJer. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, 1976.
[225] Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem.* John Wiley & Sons, 1985.
[226] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers,* EC-10(3):346–365, 1961.
[227] Tom Leighton. Tight bounds an the complexity of parallel sorting. *IEEE Transactions on Computers,* C-34(4):344–354, 1985.
[228] Tom Leighton. Notes an better master theorems for divide-and-conquer recurrences. Class notes. Available at http://citeseer.ist.psu.edu/252350. html, October 1996.
[229] Tom Leighton and Satish Rao. Multicommodity max-fiow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM,* 46(6):787–832, 1999.
[230] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. *MSDN Magazine,* October 2007.
[231] Debra A. LeJewer and Daniel S. Hirschberg. Data campression. *ACMComputing Surveys,* 19(3):261– 296, 1987.
[232] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve,* volume 1554 of *Lecture Notes in Mathematics,* pages 11–42. Springer, 1993.
[233] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics,* 126(3):649–673, 1987.
[234] L. A. Levin. Universal sorting problems. *Problemy Peredachi Iriformatsii,* 9(3):265–266, 1973. In Russian.
[235] Anany Levitin. *Introduction to the Design & Analysis of Algorithms.* Addison-Wesley, 2007.
[236] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, second edition, 1998.
[237] C. L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill, 1968.
[238] László Lovász. On the ratio of optimal integral and fractional cavers. *Discrete Mathematics,* 13(4):383– 390, 1975.
[239] László Lovász and M. D. Plummer. *Matching Theory,* volume 121 of *Annals of Discrete Mathematics.* North Holland, 1986.
[240] Bruce M. Maggs and Serge A. Plotkin. Minimum-cost spanning tree as a path-finding probJem. *Information Processing Letters,* 26(6):291–293, 1988.
[241] Michael Main. *Data Structures and Other Objects Using Java.* Addison-Wesley, 1999.
[242] Udi Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, 1989.
[243] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM,* 45(2):288–323, 1998.

[244] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences,* 20(1):18–31, 1980.

[245] H. A. Maurer, Th. Ottmann, and H.-W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters,* 5(1):11– 14, 1976.

[246] Ernst W. Mayr, Hans Jürgen Prõmel, and Angelika Steger, editors. *Lectures on Proof Verification and Approximation Algorithms,* volume 1367 of *Lecture Notes in Computer Science.* Springer, 1998.

[247] C. C. MeGeoeh. All pairs shortest paths and the essential subgraph. *Algorithmica,* 13(5):426–441, 1995.

[248] M. D. MeIlroy. A killer adversary for quieksort. *Software-Practice and Experience,* 29(4):341–344, 1999.

[249] Kurt Mehlhorn. *Sorting and Searching,* volume 1 of *Data Structures and Algorithms.* Springer, 1984.

[250] Kurt Mehlhorn. *Graph Algorithms and NP-Comple- teness,* volume 2 of *Data Structures and AIgorithms.* Springer, 1984.

[251] Kurt Mehlhorn. *Multidimensional Searching and Computational Geometry,* volume 3 of *Data Structures and AIgorithms.* Springer, 1984.

[252] Kurt Mehlhorn and Stefan Näher. Bounded ordered dietionaries in $O(\log\log N)$ time and $O(n)$ spaee. *Information Processing Letters,* 35(4):183–189, 1990.

[253] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[254] Alfred J. Menezes, Paul C. van Oorsehot, and Seott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[255] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences,* 13(3):300–317, 1976.

[256] John C. Mitehell. *Foundations for Programming Languages.* The MIT Press, 1996.

[257] Joseph S. B. Mitehell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation seheme for geometrie TSP, *k-MST,* and related problems. *SIAM Journal on Computing,* 28(4):1298–1309, 1999.

[258] Louis Monier. *Algorithmes de Factorisation D'Entiers.* PhD thesis, L'Université Paris-Sud, 1980.

[259] Louis Monier. Evaluation and eomparison of two efficient probabilistie primality testing algorithms. *Theoretical Computer Science,* 12(1):97–108, 1980.

[260] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching,* pages 285–292. Harvard University Press, 1959.

[261] Rajeev Motwani, Joseph (Seffi) Naor, and Prabakhar Raghavan. Randomized approximation algorithms in eombinatorial optimization. In Dorit Hoehbaum, editor, *Approximation Algorithms for NP-Hard Problems,* ehapter 11, pages 447–481. PWS Publishing Company, 1997.

[262] Rajeev Motwani and Prabhakar Raghavan. *Randomized AIgorithms.* Cambridge University Press, 1995.

[263] J. LMunro and V. Raman. Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica,* 16(2):151– 160, 1996.

[264] J. Nievergelt and E. M. Reingold. Binary seareh trees of bounded balance. *SIAM Journal on Computing,* 2(1):33–43, 1973.

[265] Ivan Niven and Herbert S. Zuckerman. *An Introduction to the Theory of Numbers.* John Wiley & Sons, fourth edition, 1980.

[266] Alan V. Oppenheim and Ronald W. Schafer, with John R. Buck. *Discrete-Time Signal Processing.* Prentice Hall, second edition, 1998.

[267] Alan V. Oppenheim and Alan S. Willsky, with S. Hamid Nawab. *Signals and Systems.* Prentice Hall, second edition, 1997.

[268] James B. Orlin. A polynomial time primaI network simp1ex algorithm for minimum cost flows. *Mathematical Programming,* 78(1):109–129, 1997.

[269] Joseph O'Rourke. *Computational Geometry in* C. Cambridge University Press, second edition, 1998.

[270] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[271] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, 1982.

[272] Michael S. Paterson. Progress in selection. In *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory,* pages 368–379, 1996.

[273] Mihai Pãtrascu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing,* pages 232-240, 2006.

[274] Mihai Pãtrascu and Mikkel Thorup. Randomization does not help searching predecessors. In *Proceedings of the 18th ACM-S1AM Symposium on Discrete Algorithms,* pages 555–564, 2007.

[275] Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach.* The MIT Press, 2000.

[276] Steven Phillips and Jeffery Westbrook. Online load balancing and network flow. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing,* pages 402–411, 1993.

[277] J. M. Pollard. A Monte Carlo method for factorization. *BIT,* 15(3):331–334, 1975.

[278] J. M. Pollard. Factoring with cubic integers. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve,* volume 1554 of *Lectu- re Notes in Mathematics,* pages 4–10. Springer, 1993.

[279] Carl Pomerance. On the distribution of pseudoprimes. *Mathematics of Computation,* 37(156):587– 593, 1981.

[280] Carl Pomerance, editor. *Proceedings of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography.* American Mathematical Society, 1990.

[281] William K. Pratt. *Digital1mage Processing.* John Wiley & Sons, fourth edition, 2007.

[282] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An lntroduction.* Springer, 1985.

[283] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing.* Cambridge University Press, second edition, 2002.

[284] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, third edition, 2007.

[285] R. C. Primo Shortest connection networks and some generalizations. *Bell System Technical Journal,* 36(6):1389–1401, 1957.

[286] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM,* 33(6):668–676, 1990.

[287] Paul W. Purdom, Jr. and Cynthia A. Brown. *The Analysis of Algorithms.* Holt, Rinehart, and Winston, 1985.

[288] Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results,* pages 21–39. Academic Press, 1976.

[289] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory,* 12(1):128– 138, 1980.

[290] P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica,* 7(4):365–374, 1987.

[291] Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News,* 28(2): 81–87, 1997.

[292] James Reinders. *Intel Threading Building Blocks: Outfitting c++ for Multi-core Processor Parallelism.* O'Reilly Media, Inc., 2007.

[293] Edward M. Reingold, Jürg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice Hall, 1977.

[294] Edward M. Reingold, Kenneth J. Urban, and David Gries. K-M-P string matching revisited. *Information Processing Letters,* 64(5):217–223, 1997.

[295] Hans Riesel. *Prime Numbers and Computer Methods for Factorization,* volume 126 of *Progress in Mathematics.* Birkhãuser, second edition, 1994.

[296] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM,* 21(2):120–126, 1978. See also US. Patent 4,405,829.

[297] Herbert Robbins. A remark on Stirling's formula. *American Mathematical Monthly,* 62(1):26–29, 1955.

[298] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing,* 6(3):563–581, 1977.

[299] Salvador Roura. An improved master theorem for divide-and-conquer recurrences. In *Proceedings of Automata, Languages and Programming, 24th lnternational Colloquium, ICALP'97,* volume 1256 of *Lecture Notes in Computer Science,* pages 449–459. Springer, 1997.

[300] Y. A. Rozanov. *Probability Theory: A Concise Course.* Dover, 1969.

[301] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM,* 23(3):555–565, 1976.

[302] A. Schonhage, M. Paterson, and N. Pippenger. Finding the mediano *Journal of Computer and System Sciences,* 13(2):184–199, 1976.

[303] Alexander Schrijver. *Theory of Linear and 1nteger Programming.* John Wiley & Sons, 1986.

[304] Alexander Schrijver. Paths and flows-A historical survey. *CW1 Quarterly*, 6(3):169–183, 1993.

[305] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM,* 21(10):847–857, 1978.

[306] Robert Sedgewick. *Algorithms.* Addison-Wesley, second edition, 1988.

[307] Robert Sedgewick and Philippe Flajolet. *An 1ntroduction to the Analysis of Algorithms.* Addison-Wesley, 1996.

[308] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences,* 51(3):400–403, 1995.

[309] Raimund Seidel and C. R. Aragon. Randomized search trees. *Algorithmica,* 16(4–5):464–497, 1996.

[310] João Setubal and João Meidanis. *1ntroduction to Computational Molecular Biology.* PWS Publishing Company, 1997.

[311] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis.* Prentice Hall, second edition, 2001.

[312] Jeffrey Shallit. Origins ofthe analysis ofthe Euclidean algorithm. *Historia Mathematica,* 21(4):401–419, 1994.

[313] Michael I. Shamos and Dan Hoey. Geometric intersection problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science,* pages 208–215, 1976.

[314] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications,* 7(1):67–72, 1981.

[315] David B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. In William Cook, László Lovász, and Paul Seymour, editors, *Combinatorial Optimization,* volume 20 of *DIMACS Series in Discrete Mathematics and Theoretical Compu ter Science.* American Mathematical Society, 1995.

[316] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual Symposium on Foun- dations of Computer Science,* pages 605–614, 1999.

[317] Michael Sipser. *1ntroduction to the Theory of Com- putation.* Thomson Course Technology, second edition, 2006.

[318] Steven S. Skiena. *The Algorithm Design Manual.* Springer, second edition, 1998.

[319] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences,* 26(3): 362–391, 1983.

[320] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM,* 32(3):652–686, 1985.

[321] Joel Spencer. *Ten Lectures on the Probabilistic Method,* volume 64 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics, 1993.

[322] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM,* 51(3):385–463,2004.

[323] Gilbert Strang. *lntroduction to Applied Mathematics.* Wellesley-Cambridge Press, 1986.

[324] Gilbert Strang. *Linear Algebra and lts Applications.* Thomson Brooks/Cole, fourth edition, 2006.
[325] Vo1ker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.
[326] T. G. Szymanski. A specia1 case of the maximal common subsequence prob1em. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975.
[327] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing,* 1(2):146–160, 1972.
[328] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM,* 22(2):215– 225, 1975.
[329] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Jour- nal of Computer and System Sciences,* 18(2):110–127, 1979.
[330] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.
[331] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods,* 6(2):306–318, 1985.
[332] Robert E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999.
[333] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM,* 31(2):245–281, 1984.
[334] George B. Thomas, Jr., Maurice D. Weir, Joe1 Hass, and Frank R. Giordano. *Thomas' Calculus.* Addison-Wes1ey, eleventh edition, 2005.
[335] Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms,* pages 550–555, 1998.
[336] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM,* 46(3):362–394, 1999.
[337] Mikke1 Thorup. On RAM priority queues. *SIAM Journal on Computing,* 30(1):86–109, 2000.
[338] Richard To1imieri, Myoung An, and Chao Lu. *Mathematics of Multidimensional Fourier Transform Algorithms.* Springer, second edition, 1997.
[339] P. van Emde Boas. Preserving order in a forest in 1ess than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science,* pages 75–84, 1975.
[340] P. van Emde Boas. Preserving order in a forest in 1ess than logarithmic time and linear space. *lnformation Processing Letters,* 6(3):80–82, 1977.
[341] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory,* 10(1):99–127, 1976.
[342] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity.* EIsevier Science Pub1ishers and the MIT Press, 1990.
[343] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform.* Society for Industrial and Applied Mathematics, 1992.
[344] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions.* Kluwer Academic Publishers, 1996.
[345] Vijay V. Vazirani. *Approximation AIgorithms.* Springer, 2001.
[346] Rakesh M. Verma. General techniques for analyzing recursive algorithms with applications. *SIAM Journal on Computing,* 26(2):568–581, 1997.
[347] Hao Wang and Bill Lin. Pipelined van Emde Boas tree: Algorithms, analysis, and applications. In *26th IEEE International Conference on Computer Communications,* pages 2471–2475, 2007.
[348] Antony F. Ware. Fast approximate Fourier transforms for irregularly spaced data. *SIAM Review,* 40(4):838–856, 1998.
[349] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM,* 9(1):11–12, 1962.
[350] Michael S. Waterman. *Introduction to Computational Biology, Maps, Sequences and Genomes.* Chapman & Hall, 1995.
[351] Mark Allen Weiss. *Data Structures and Problem Solving Using C++.* Addison-Wesley, second edition, 2000.
[352] Mark Allen Weiss. *Data Structures and Problem Solving Using Java.* Addison-Wesley, third edition, 2006.
[353] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++.* Addison-Wesley, third edition, 2007.
[354] Mark Allen Weiss. *Data Structures and Algorithm Analy- sis in Java.* Addison-Wesley, second edition, 2007.
[355] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics,* 57(3):509–533, 1935.
[356] Herbert S. Wilf. *AIgorithms and Complexity.* A K Peters, second edition, 2002.
[357] J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM,* 7(6):347–348, 1964.
[358] Shmuel Winograd. On the algebraic complexity of functions. In *Actes du Congres International des Mathématiciens,* volume 3, pages 283–288, 1970.
[359] Andrew c.- C. Yao. A lower bound to finding convex hulls. *Journal of the ACM*, 28(4):780–787, 1981.
[360] Chee Yap. A real elementary approach to the master recurrence and generalizations. Unpublished manuscript. Available at http://cs.nyu.edu/yap/ papers/, July 2008.
[361] Yinyu Ye. *Interior Point Algorithms: Theory and Analysis.* John Wiley & Sons, 1997.
[362] Daniel Zwillinger, editor. *CRC Standard Mathematical Tables and Formulae.* Chapman & Hall/CRC Press, 31 st edition, 2003.

ÍNDICE REMISSIVO

Símbolos

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W